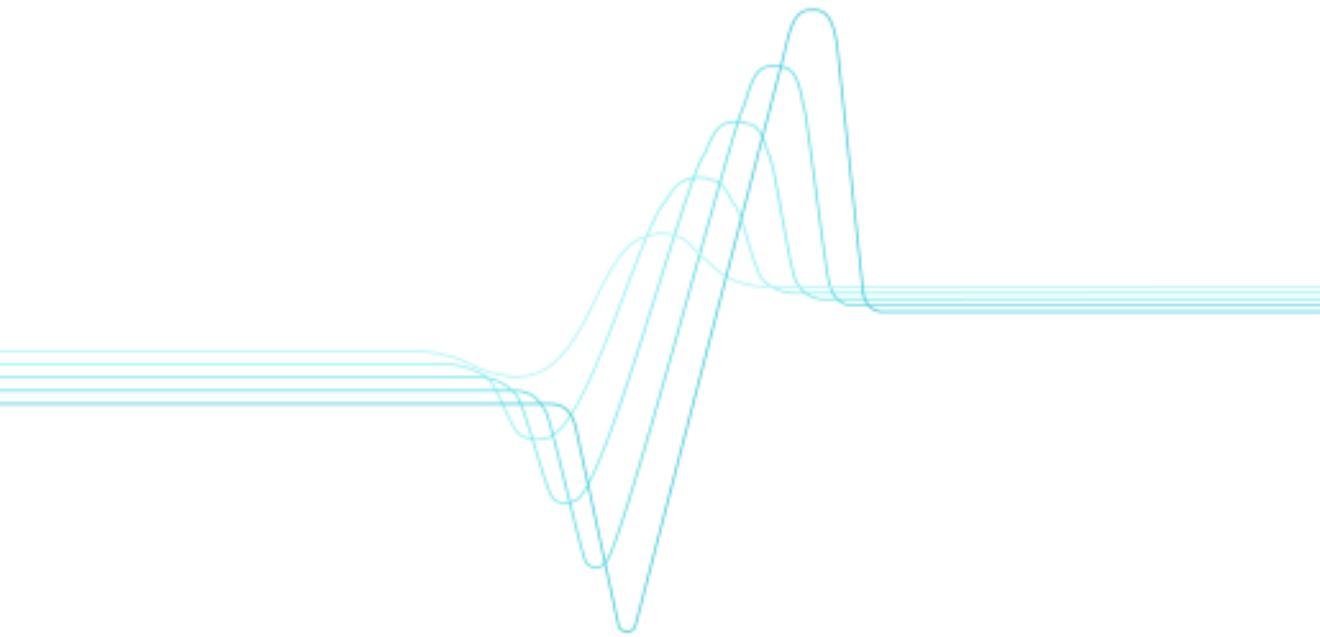


Juho Jäälinoja

Requirements implementation in embedded software development



VTT PUBLICATIONS 526

Requirements implementation in embedded software development

Juho Jäälinoja

VTT Electronics



ISBN 951-38-6370-0 (URL: <http://www.vtt.fi/inf/pdf/>)

ISSN 1455-0849 (URL: <http://www.vtt.fi/inf/pdf/>)

Copyright © VTT Technical Research Centre of Finland 2004

JULKAISIJA – UTGIVARE – PUBLISHER

VTT, Vuorimiehentie 5, PL 2000, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 4374

VTT, Bergsmansvägen 5, PB 2000, 02044 VTT
tel. växel (09) 4561, fax (09) 456 4374

VTT Technical Research Centre of Finland, Vuorimiehentie 5, P.O.Box 2000, FIN-02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 4374

VTT Elektronikka, Kaitoväylä 1, PL 1100, 90571 OULU
puh. vaihde (08) 551 2111, faksi (08) 551 2320

VTT Elektronik, Kaitoväylä 1, PB 1100, 90571 ULEÅBORG
tel. växel (08) 551 2111, fax (08) 551 2320

VTT Electronics, Kaitoväylä 1, P.O.Box 1100, FIN-90571 OULU, Finland
phone internat. + 358 8 551 2111, fax + 358 8 551 2320

Technical editing Marja Kettunen

Jäälinoja, Juho. Requirements implementation in embedded software development. Espoo 2004. VTT Publications 526. 82 p. + app. 7 p.

Keywords software process improvement, software requirements analysis, embedded systems

Abstract

Development of correct requirements at the beginning of a software project is considered an important precondition for successful software development. Moreover, implementing these requirements correctly during the software development is arguably just as important. Rigorous implementation of requirements in embedded software development is especially critical, since requirements affect both software and hardware. The goal of this research is to identify elements for effective requirements implementation in embedded software development.

A conceptual-theoretical research approach is applied to analyse previous research on requirements implementation and to construct a new theory which integrates requirements implementation related elements into a holistic framework. These elements include requirements implementation processes, methods, and roles. The developed framework describes relations among these elements and furthermore their relation to software development activities. The framework can be used as a basis for improving software development areas that are related to requirements implementation.

To validate the feasibility of the developed framework, two case studies were carried out within embedded software development organisations. The validation was conducted by making a current state analysis and by suggesting improvements based on the developed requirements implementation framework. The results from the case studies indicated that the framework was a useful foundation for improving the organisations' requirements implementation practices.

Preface

This research was conducted at VTT Electronics in the MOOSE (software engineering MethodOlogieS for Embedded systems) project. The project's goal was to improve the integration of available tools, techniques, methods, and processes involved in embedded software development. The project started at the beginning of 2002 and I joined the project a year later. In the project I had an opportunity to study and write my graduate thesis on requirements and embedded software development. This publication is based on that thesis. I must sincerely thank VTT Electronics and other partners in the project for allowing me to write the thesis as part of the project.

I wish to thank all the people at VTT Electronics for their support. Especially, I would like to thank Ms. Päivi Parviainen for competent guidance throughout the study. Furthermore, I want to thank Prof. Markku Oivo from the University of Oulu for his support. I am also grateful for the co-operative people in the two anonymous organisations I had the opportunity to study.

Oulu, March 2004

Juho Jäälinoja

Contents

Abstract.....	3
Preface	4
Abbreviations.....	7
1. Introduction.....	9
1.1 Background.....	9
1.2 The research problem and methods	10
1.3 Scope	12
1.4 Structure	12
2. Embedded systems and requirements	14
2.1 Embedded systems	14
2.2 Embedded systems development.....	15
2.3 Embedded software development.....	16
2.4 Requirements.....	18
2.5 Requirements engineering	21
3. Requirements in software development activities	24
3.1 Software requirements analysis.....	24
3.2 Requirements and software architecture design	26
3.3 Requirements and detailed software design	29
3.4 Requirements and software coding.....	30
3.5 Requirements and software testing.....	31
4. Requirements implementation-supporting elements.....	34
4.1 Requirements change management	34
4.1.1 Impact analysis.....	36
4.1.2 Change management tools	37
4.2 Requirements tracing.....	38
4.3 Consistency management	41
5. Requirements implementation framework.....	45
5.1 Requirements implementation framework for embedded software.....	45

5.1.1	Requirements implementation in software requirements analysis.....	46
5.1.2	Requirements implementation in software designing	50
5.1.3	Requirements implementation in software coding.....	52
5.1.4	Requirements implementation in software testing	54
5.1.5	Requirements implementation throughout the development...	56
5.2	Adaptation of the framework.....	59
6.	Validation of the requirements implementation framework	62
6.1	Research approach for validation	62
6.2	Case one - large organisation.....	63
6.2.1	Current state analysis	64
6.2.2	Improvement proposal	65
6.3	Case two - small organisation.....	67
6.3.1	Current state analysis	67
6.3.2	Improvement proposal	68
6.4	Applicability of the requirements implementation framework.....	69
7.	Conclusions.....	71
7.1	Answers to the research questions.....	71
7.2	Significance of the results	72
7.3	Further research possibilities	74
	References.....	75

Appendices

Appendix 1: Questionnaire on current state requirements implementation practices

Appendix 2: Requirements implementation practices

Abbreviations

AQA	Architecture Quality Analysis
ASIC	Application-Specific Integrated Circuit
ATAM	Architecture Tradeoff Analysis Method
C BSP	Component-Bus-System-Property
CCB	Change Control Board
CMM	Capability Maturity Model
CMMI	Capability Maturity Model Integration
DOORS	Dynamic Object-Oriented Requirements System
DSP	Digital Signal Processing
GRL	Goal-oriented Requirement Language
MOOSE	software engineering Methodologies for Embedded systems
PDL	Program Design Language
QFD	Quality Function Deployment
RTM	Requirements Traceability Management
SBAR	Scenario-Based Architecture Reengineering
SCM	Software Configuration Management
SCR	Software Cost Reduction
SDL	Specification and Description Language
SRS	Software Requirements Specification
TCM	Test Case Management
UCM	Use Case Maps
UML	Unified Modeling Language
VTT	Technical Research Centre of Finland (Valtion teknillinen tutkimuskeskus)

1. Introduction

Modern embedded systems are more complex than before, and much of this complexity is realised in these systems' software. Organisations that neglect to manage software development efficiently are doomed to fail in these complex development projects. Developing correct requirements and implementing them correctly are considered key elements in successful software development. In this research, the latter condition is studied and related means for improving requirements implementation are provided. The term *requirements implementation* refers in this research not only to realising requirements in code, but also to their elaboration into software requirements, designs, and test cases. This chapter presents the background for the study, defines the research problem and scope, and outlines the structure of the study.

1.1 Background

This study was carried out at VTT Electronics within the MOOSE (software engineering MethodOIOgieS for Embedded systems) project, which aimed to improve the integration of available tools, techniques, methods, and processes involved in embedded software development. System and software requirements engineering are among the research areas in the project. The industrial partners of the project expressed several problems related to requirements. The following list contains some examples (MOOSE 2002b):

- requirements are hard to specify and difficult to know up-front
- software expertise is absent from system requirements allocation
- change of requirements creates inconsistency amongst work products
- tracing of requirements throughout the development is difficult

One of the main problems of requirements implementation is the inevitable change of original requirements. Requirements may be removed, modified, or added to during the development. Software development is very rarely a linear front-end process, where requirements can be agreed upon at the beginning and the rest of the development will be based on them. Instead, software typically

iterates towards greater maturity. During the iterations original requirements very often change, which can cause problems for design, implementation, and testing activities. Problems typically arise when a change of one requirement affects many different components in the system.

Correct implementation of requirements in embedded software development is a critical success factor for the whole system development since it affects both software and hardware. For instance, an inconsistency between requirements and software designs leads to incorrect implementation of the software and furthermore might trigger unexpected behaviour in the system's hardware. The later a defect is found in the system development, the more it will cost to repair.

This research explores solutions to these problems from the point of view of requirements implementation. Although there has been made a significant number of research on development, implementation, management, and consistency management of requirements, these studies unfortunately present only insular solutions. A holistic view of these elements is needed in order to locate essential improvement areas in software development. Furthermore, current research lacks an embedded software viewpoint regarding these issues. It is obvious that concurrent development of hardware and software creates specific problems for requirements implementation. In this study, the previously stated deficiencies in current research are addressed.

1.2 The research problem and methods

The purpose of this research is to study effective implementation of requirements in embedded software development. The research problem is stated as the following research question and sub-questions:

1. How can requirements be effectively implemented in embedded software development?
 - 1.1. What is the relationship between requirements and development activities?
 - 1.2. How should requirements be implemented during development?

1.3. What kind of framework would help to analyse and improve requirements implementation practices?

The research problem is solved with conceptual-theoretical and theory-testing research approaches. Conceptual-theoretical research approach usually includes an analysis phase, which is sometimes followed by a synthesis phase. The objective of the analysis phase is to study how previous research has structured the subject of research. (Järvinen & Järvinen 2000.) The first research sub-question is answered here in the analysis phase by studying the relationship between requirements and development activities based on current literature. Literature is also used as a source for answering the second sub-question. Elements for effective requirements implementation are assembled from software engineering standards, books and research papers.

The synthesis phase of conceptual-theoretical research approach allows us to construct a new model or a theory, which describes the subject of the research more accurately (Järvinen & Järvinen 2000). In the synthesis phase, the third research sub-question is answered by integrating requirements implementation related elements found in the analysis phase into a holistic framework. The developed framework describes relations among requirements implementation elements and furthermore their relation to software development activities. The utilisation of the framework should not oblige us to use any specific methods or techniques; instead, it suggests what kinds of techniques can be used in rigorous implementation of requirements and how these techniques can be integrated. The framework can be used as a basis for improving software development areas that are related to requirements implementation.

A theory-testing research approach is applied to validate the usability of the developed framework in an industrial environment. The validation includes two case studies within embedded software development organisations. First, a current state analysis of an organisation's requirements implementation practices in software development process is carried out with an historical experimentation research approach according to Zelkowitz and Wallace (1998). After the analysis, an improvement proposal is composed based on a comparison between the current state and the developed framework. Finally, the proposal is evaluated by the case organisation to get feedback on the applicability of the framework.

1.3 Scope

This research concerns both requirements engineering and software implementation. Requirements engineering gives input to software implementation and pervades its management elements over the whole software development life cycle. On the other hand, there is a significant flow of information from software implementation to requirements development activities. For example, requirements are very hard to know up-front, and therefore they tend to refine as the software matures. In this research, related areas of requirements engineering and software implementation are covered and their interrelations are studied.

The research is not merely concerned about methods that elaborate requirements into software analysis models, designs, source code and test cases. Neither does the research concentrate purely on the requirements management issues. Instead, the scope of the research is to study rigorous implementation of requirements, consistency management between requirements and software work products, and how requirements and change management can be integrated into software development activities.

The research is restricted to study requirements implementation in embedded software development context. Many of this research's concepts, however, are applicable for any kind of software development. The requirements implementation framework's usefulness is validated in embedded software environment with industrial case studies by making a current state analysis and suggesting improvement proposals. However, implementing the proposed improvements and then evaluating the possible improvement of the software process is not within the scope of this research.

1.4 Structure

The structure of this research is illustrated in Figure 1. Chapter 2 introduces the basic concepts of the study. Special characteristics of embedded systems and embedded software development are discussed. Furthermore, definitions for requirements and requirements engineering are given.

Chapter 3 examines the relationship between requirements software development activities such as requirements analysis, design, coding, and testing. Also elements needed for effective implementation of requirements in the development activities are presented.

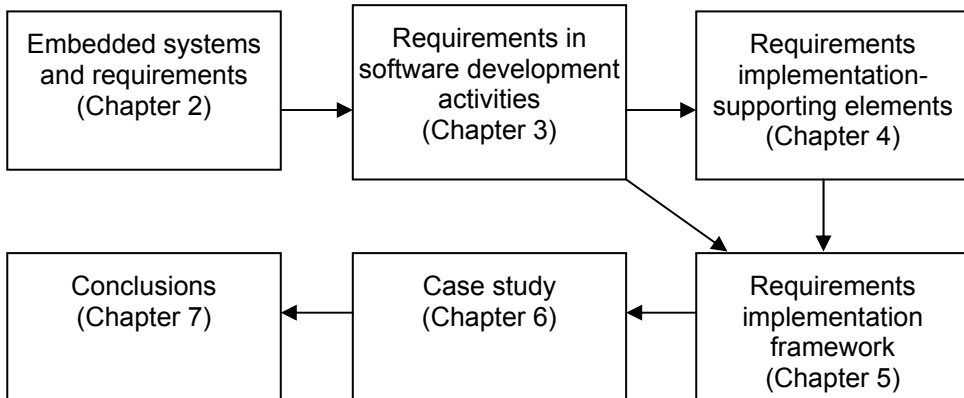


Figure 1. Structure of the research

Chapter 4 presents requirements implementation-supporting elements which are used throughout the software development. These elements include change management, traceability of requirements, and consistency management between requirements and different software work products. The first and second research sub-questions are answered in Chapters 3 and 4.

Chapter 5 answers the third research sub-question by presenting a framework that integrates elements of effective requirements implementation discussed previously in Chapters 3 and 4. The sub-question is further handled in Chapter 6, where the validation of the framework is discussed. Two case studies that validate the framework are presented and the most important findings are discussed.

The last chapter concludes the research and discusses the results. Further study opportunities based on this research are also given. Appendix 1 contains a questionnaire that can be used to clarify an organisation's current state of requirements implementation practices. A summary of requirements implementation practices and methods referenced to in this study is given in Appendix 2.

2. Embedded systems and requirements

This chapter introduces the basic concepts of the research. Because the focus of this research is requirements implementation in embedded software domain, special characteristics of embedded systems, embedded systems development, and embedded software development have to be taken into account. Furthermore, definitions for requirements and requirements engineering are given.

2.1 Embedded systems

Embedded systems are electro-mechanical products which relate mechanics, hardware and software. Examples of embedded systems are mobile phones, medical instruments and petrol dispensers. (Taramaa et al. 1998.) An embedded system contains a computer which is part of a larger system that provides non-computing features to the user (Douglass 2000). The size of an embedded system might vary from a small thermometer to a chemical plant's process controller.

Generally, an embedded system is composed of components implementing different hardware and software functions. Hardware functions may be implemented by off-the-shelf components, programmable logic and Application-Specific Integrated Circuits (ASICs). Software functions may be implemented by off-the-shelf processors such as Digital Signal Processors (DSPs) and other specific processors. (López et al. 1998.) Embedded systems typically contain a small microcontroller and limited software. However, with the development of such systems more sophisticated microcontrollers, DSP chips, off-the-shelf real-time operating systems, and software are being used. (Stankovic 1996; López et al. 1998.)

Embedded systems that have timing constraints are called real-time systems, which not only have to produce correct output, but also should produce it at the right time. Embedded real-time systems often operate in tight timing constraints, where missing an important deadline can lead to severe consequences. (Stankovic 1996.) Some real-time embedded systems operate in a safety-critical domain, where functionality and reliability are critically important features. Examples of safety-critical real-time embedded systems are avionics, traffic control systems, and medical devices.

2.2 Embedded systems development

The development of embedded systems is difficult, because these systems are part of a physical environment whose complex dynamics and timing requirements they must pursue. Taramaa et al. (1998, p. 907) define other key characteristics of embedded systems development:

- embedded systems design is often constrained by the implementation technology
- different technologies used in the same system can be developed simultaneously
- the system can possess both low-level and high-level software implementation
- maintainability and extensibility of embedded systems is achieved with new functions, technologies, and interfaces
- short development time

Although development of embedded systems has been a common practice since the first microprocessors, traditional development approaches are no longer valid to ensure high quality under strict time-to-market and design cost constraints. One of the reasons for this is the increasing complexity of today's microprocessors and associated real time software. Also the complexity of hardware and software interface communication mechanisms increases the need for more suitable development approaches. (López et al. 1998.)

As a solution, more sophisticated development approaches for such complex embedded systems are being developed. These methodologies are called hardware/software co-design methodologies. (López et al. 1998.) The traditional sequential development approach has been replaced by concurrent development of hardware and software in the co-design approach. Hardware and software design activities may begin with immature system specifications and architectural designs. Concurrent development with incomplete specifications requires close collaboration from all participants. For example, interfaces between software and hardware must be designed in co-operation by both hardware and software designers. Also testing processes for hardware and

software have to be integrated. Reusing components from previous designs or outside the design group is considered as the main design goal. (Ernst 1998.) Embedded systems co-design process is shown in Figure 2.

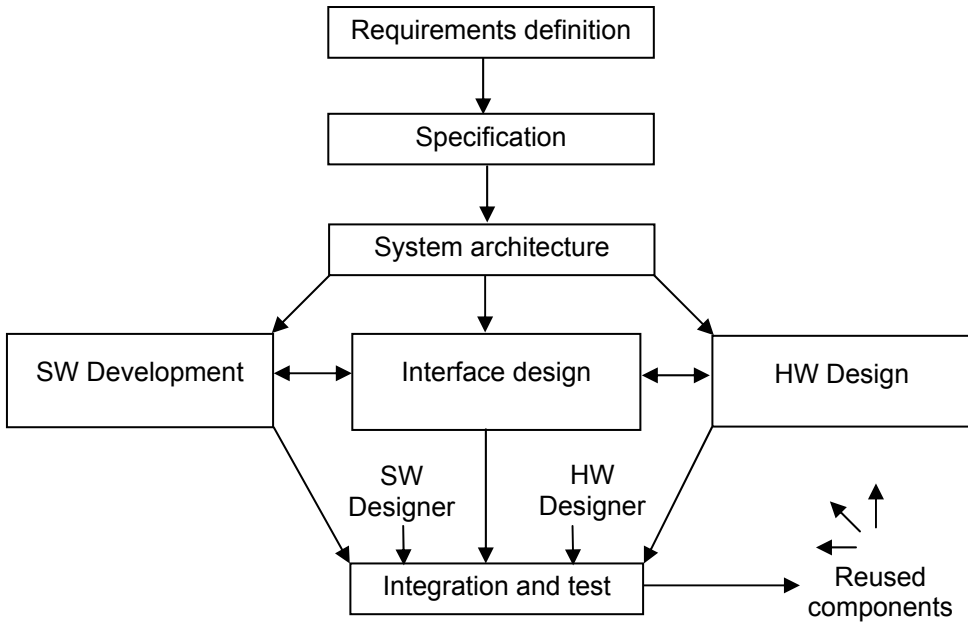


Figure 2. Embedded systems design process (adapted from Ernst 1998, 46).

The functionality of an embedded system is usually fixed and is mainly determined by its interaction with the system's environment. An embedded system often has numerous operation modes, capability to respond to exceptions, and features that demand concurrent execution of different operations. (Gajski & Vahid 1995.) These functions are handled by the embedded system's built-in software, which is closely connected to the hardware. The close connection between software and hardware means that the software development is often steered and restricted by the hardware. (Lee 2000.)

2.3 Embedded software development

Software development is one component of the whole embedded systems development process. Embedded systems development from the viewpoint of

software is presented as the classical V-model according to Easterbrook (2001) and the terminology of ISO/IEC 12207 in Figure 3. This model's concepts are used throughout this research. However, system level analysis, design, and testing activities are not in the scope of the research.

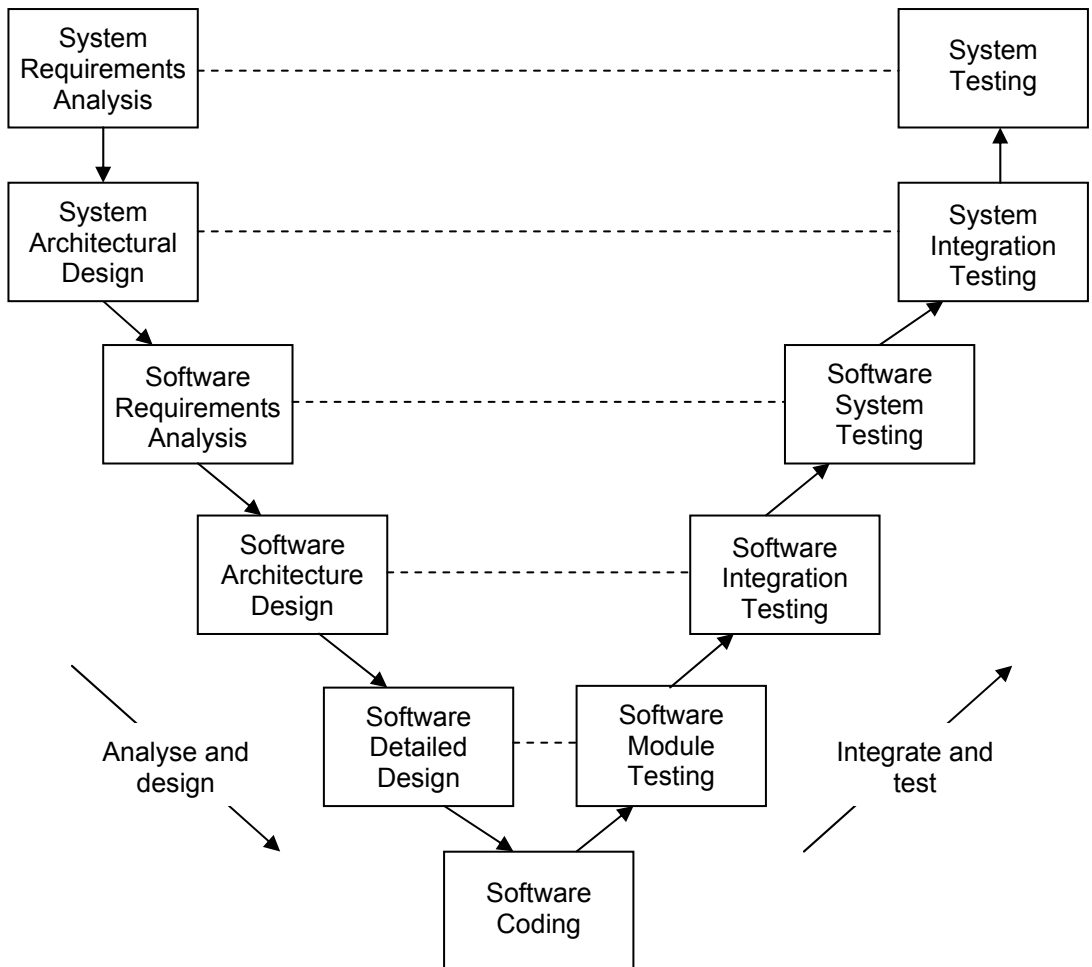


Figure 3. Software development V-model (based on Easterbrook 2001 and ISO/IEC 12207).

The V-model basically states that the system requirements on the left side of the V are gradually refined in the analysis and design activities until they are implemented as software code. The activities on the right side of the V then integrate and test the software until it is ready to be integrated with other

possible systems and hardware. The dotted lines between the development activities indicate that the testing activities are based on the analysis and design activities. Although verification and validation of the product is closely related to testing, all the work products from the development activities must be verified and validated throughout the development. Verification means checking that an activity's work products fulfil the requirements or conditions imposed on them in the previous activities. Validation ensures that the requirements and the final product fulfil the product's specific intended use. (ISO/IEC 12207.)

It should be understood that the V-model is a simplified representation of software development. There is a constant information flow from lower-level development activities such as coding and design to higher-level requirements analysis and design activities. The development activities are not separate entities performed by different people; on the contrary, there is a lot of overlap and collaboration among these activities and system and hardware development activities. Furthermore, the software development activities interrelations and performance order may vary.

Programmers write today's embedded software using low-level programming languages, such as C or even assembly language, to cope with constraints on performance and cost. Object-oriented languages such as C++ are not as popular. (MOOSE 2002b.) Implementation and debugging tools for embedded software are basically the same as those for traditional software, such as compilers, assemblers, and debuggers. However, most tools for embedded software development are immature compared to traditional software development tools. (Sangiovanni-Vincentelli & Martin 2001.)

2.4 Requirements

A requirement is something that the system must exhibit or a quality that the system must have. Requirements are defined at the beginning of the development as a specification of what should be developed. Requirements for a system are typically a combination of demands from different people at different levels of the developing organisation and from the environment where the system must operate (Sawyer & Kotonya 2001, 2). In addition to the description of what the system should do, requirements often involuntarily describe how the



system should be done. Therefore, requirements are a mixture of problem information, statement of behaviour, and design and manufacturing constraints. (Sommerville & Sawyer 1997.)

Requirements used in the development of embedded systems are generally classified as system, hardware, and software requirements. This research concerns itself mostly with system and software requirements. System requirements are derived from various high-level requirements stated by different stakeholders. These stakeholders are for example acquirers, end-users, sales and marketing people, regulators, system developers, and other existing systems. System requirements are expressed in the terms of the stakeholders' domain and are usually documented in a system requirements specification (Sawyer & Kotonya 2001).

Software requirements are detailed requirements for software derived from the system requirements. Software requirements fall into two general categories: functional and non-functional requirements. Functional requirements define capability and behaviour of the system. A sample functional requirement would be a system's capability to format some text or modulate a signal. (Sawyer & Kotonya 2001.) Non-functional requirements refer to performance, interface, quality, and design constraints of the software. For example, a non-functional requirement could define a recovery time for the software or specify the programming language used in the implementation of the software. Software requirements categorisation based on IEEE 830-1998 standard with explanatory questions is listed in Table 1.

Table 1. Categorisation of software requirements.

Functional requirements	What is the software supposed to do?
Performance requirements	What are availability, response time, and recovery time of software functions?
External interface requirements	How does the software interact with external entities, such as hardware?
Quality attributes	What are the quality considerations of software such as reliability, usability, maintainability, and portability?
Design constraints	Are there any required standards in effect, implementation languages, operating environments, etc.?

 Non-functional requirements
 Functional requirements

In order to achieve quality in the developed product, it is not enough only to fulfil the functional requirements. When a non-functional requirement such as usability is neglected, the resulting system might work correctly but it may be difficult to use. Unfortunately, non-functional requirements are often not as easily verifiable as functional requirements during the development. (Ebert 1997.)

Functional and non-functional requirements for software are typically documented in a software requirements specification (SRS). The readers of an SRS are expected to have knowledge of software engineering concepts, which can reflect on the language and notation used in the specification. (Sawyer & Kotonya 2001.) In addition to the software requirements that are stated in natural language, graphical representations are often needed to depict a full picture of the intended system. These representations as analysis models also help developers to detect inconsistencies, errors, and omissions in the requirements. Furthermore, graphical descriptions improve developers understanding of the requirements. (Wiegiers 1999.) The SRS and its graphical representations are discussed further in the next chapter.

Requirements for embedded software have a different emphasis than requirements for conventional desktop software. In conventional software development, non-functional requirements such as timing, reliability, robustness, and power consumption are secondary to the logical correctness of computations. In embedded software development, non-functional requirements

are equally important as functional requirements. (Karsai et al. 2003.) Non-functional requirements are especially recognised as an important success factor in market-driven embedded software development (Punter et al. 2002). A product with outstanding functionality is not enough; instead, both functionality and quality characteristics are needed for a successful product.

In addition, system and hardware requirements are more important in embedded software development than in development of conventional software. During the system design, an agreement based on system requirements is made about the system's components and their purposes. Then, software and hardware requirements are elaborated for each of these components separately. (Davis 1990.) Therefore, development of embedded software is dependent on system and hardware requirements. Developers of conventional software hardly need to worry about system level requirements (except for large software systems) and even less so about hardware requirements.

2.5 Requirements engineering

The term requirements engineering has various definitions. Robinson and Pawlowski (1999) characterise requirements engineering as an iterative process of discovery and analysis of agreed set of clear, complete, and consistent system requirements. This research shares a broader perspective on requirements engineering, including the management viewpoint, as stated by Dorfman (1997, p. 12): "...requirements engineering consists of elicitation, analysis, specification, validation/verification, and management [of requirements]."

Requirements engineering can be further classified into system and software requirements engineering. System requirements engineering is concerned with analysing and documenting system requirements. Requirements engineering from systems engineering viewpoint is shown in Figure 4. In system requirements engineering, stakeholders' needs are transformed into system description, system performance parameters, and system configuration. System-level requirements engineering is also concerned with the partitioning of the system. This is achieved by identifying which requirements should be allocated to which components (e.g. software, hardware, or subsystem). (Sawyer & Kotonya 2001.) The allocation is based on different criteria such as cost of

implementation, cost of replication, implementation time, ease of change, and error rates during operation (Stevens et al. 1998).

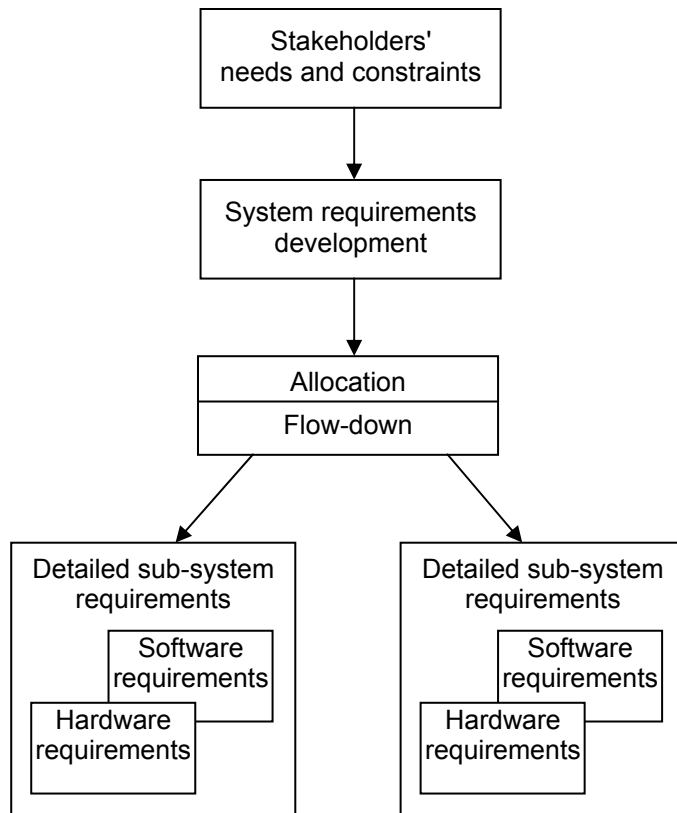


Figure 4. Requirements engineering in systems engineering context (adapted from Parviainen et al. 2003, 13).

After the allocation, a flow-down process will take place, where requirements for the lower-level elements are specified in response to the allocation. The lower-level elements must have at least one requirement that responds to the allocation, but often many requirements are written for one allocation. (Dorfman 1997.) Finally, requirements are implemented in sub-systems either in hardware or software disciplines.

Software requirements engineering is concerned with establishing and documenting software requirements. Allocated system requirements are transformed into software requirements through the use of analysis, design,

trade-off studies, and prototyping. The major difference between system and software requirements engineering is that the origins of system requirements are stakeholders' needs, while the origins of software requirements are system requirements. (Thayer & Dorfman 1997.) However, in small scale development, software requirements may come directly from stakeholders.

Sawyer and Kotonya (2001) have defined requirements engineering activities, which are broadly compatible with the ISO/IEC 12207 (1995) software life cycle processes standard. Requirements engineering consists of activities such as elicitation, analysis, specification, validation, management of requirements. These activities, except for requirements management, should comprise an iterative process where activities are repeated until an acceptable requirements document is produced. This process is not a separate front-end activity in the software life cycle, but rather an activity that is initiated at the beginning of the life cycle and continues to be refined through the life cycle. Requirements management is an activity that spans the entire software development. It consists of requirements identification, tracing, and change management. (Sawyer and Kotonya 2001.)

3. Requirements in software development activities

After the agreed system requirements have been specified and further allocated to software, the work begins on evolving these requirements into a software requirements specification, designs, code, and test cases. This chapter discusses the relationships between requirements and software development activities of the V-model presented in the previous chapter. In addition, elements needed for effective implementation of requirements in the development activities are presented. These elements were gathered from current literature, which includes software engineering standards, research papers, and books. The presented elements are later used in Chapter 5 to construct the requirements implementation framework.

3.1 Software requirements analysis

Software requirements analysis is a software engineering activity that brings system requirements closer to software designs. In this activity, system requirements are elaborated into software requirements. The elaboration is done by refining, analysing, modelling, and specifying allocated system requirements into software requirements. To increase knowledge about what is required from the software, the problem is partitioned and representations that describe requirements for software are constructed. Thus, requirements analysis is a technical step, where broad expressions of the software are refined into a more detailed specification that serves as a foundation for all other software engineering activities that follow. (Pressman 2000.)

Two distinct activities are performed during the specification of software requirements: problem analysis and product description. Problem analysis is an activity where analysts brainstorm ideas, interview people, and identify all possible constraints on the problem's solution. Product description activity is concerned with the external behaviour of the product. The purpose of the product description activity is to solve the problem that is understood after the problem analysis. (Davis 1990.) Possible approaches for these activities are structured analysis, object-oriented analysis, and formal methods. Sample

techniques include data flow diagrams, state-transition diagrams, and use cases. (Parviainen et al. 2003.)

A software requirements specification (SRS) is developed as a result of the requirements analysis. If necessary, it may be jointly prepared with customers, because software engineers usually do not understand customers' problems and domain area well enough to produce correct requirements (IEEE Std 830-1998). On the other hand, software developers who will later implement and test these requirements should be also included in the software requirements analysis activity. For example, software architects could prevent the acceptance of unachievable requirements. Benchmarking and prototyping the architecture before the requirements are nailed down may reveal problems with the suggested requirements. (Mead 1994.) Software testers should be included in requirements reviews to make sure that the requirements can be eventually verified. All the requirements included in the SRS should have a proper verification criterion. (Wiegiers 1999.)

Requirements in an SRS should be documented with related attributes so that they are identified and prioritised. Identification is needed for requirements management. Priorities for requirements should be assigned to define implementation order and allow resolution of conflicts in later development activities. Customers naturally demand high priorities for requirements that are important for users. In contrast, developers desire high priorities for requirements that are important for software development and maintenance. The cost of a certain requirement plays also major role in deciding what is essential to implement. Reconciliation of these different viewpoints can be supported, for example, with Quality Function Deployment (QFD) and Wiegiers' method. (Wiegiers 1999.)

A requirement's stability may be also be stated as an attribute. For some requirements it is possible to define the probability of their change in the future. The stability attribute enables software developers to construct designs that are more tolerant of change. Other possible attributes for requirements include source and scope of the requirement. (Sawer & Kotonya 2001.)

An SRS must be in line with system requirements. To achieve this, traceability between system and software requirements must be established. Furthermore,

software requirements consistency with system requirements must be ensured. (ISO/IEC 12207:1995, IEEE 830-1998.) The SRS should be comprehensive and should not include any design, implementation, testing, or project management details other than applicable design and implementation constraints. An undocumented requirement is not part of the agreement and no one should expect it to be in the product. Although all the needed requirements should be contained in the SRS, software design and construction can begin before the entire SRS is written. This requires that the development project has baselined agreement for a set of requirements. Baselined requirements are approved and reviewed requirements from the SRS under development. Changes to the baselined requirements should be made through a defined change control process. (Wiegiers 1999.)

After the software requirements have been analysed and specified, they need to be verified. Verification ensures that requirements (1) correctly describe behaviour and characteristics of the system, (2) are internally consistent, (3) are complete and high-quality, and (4) provide an adequate basis to proceed with design, implementation, and testing. Verification is not a solitary activity performed after all of the requirements have been documented. Instead, as the requirements tend to develop during the requirements gathering and analysis, they need to be verified incrementally. (Wiegiers 1999.)

One possible approach for verification of software requirements is a formal inspection of the SRS. This is especially useful when requirements are being baselined for further development of software. Reading techniques such as perspective-based reading (Shull et al. 2000) can be used in requirements inspections. In addition to inspections, some quality attributes, such as safety, might require their own specific verification procedures (Rodríguez-Dapena et al. 2001).

3.2 Requirements and software architecture design

Software's architectural design depicts a holistic view of the software to be built. It represents the structure and organisation of software components, their properties, and the connections between them. The architecture enables software engineers to analyse the effectiveness of the design in meeting its requirements.

Furthermore, risks associated with the construction of the software are reduced by comparing alternative architectural solutions. (Pressman 2000.) Having a good software architecture does not ensure that the developed product will meet its requirements. On the other hand, having a badly designed architecture makes it almost impossible to meet the product's requirements. (Hofmeister et al. 2000.)

Requirements are a key input to the design of software architecture. Elaborating requirements into valid software architecture that satisfies those requirements is a difficult task and is mainly based on intuition. Little is known about the impact of architectural choices on the requirements. (Grünbacher et al. 2001.) The architectural design must meet both functional and non-functional requirements. For example, performance requirements and timing constraints for embedded software influence architectural design. Sample constraints include data throughput rates, function call latencies, and interrupt congestion (Ronkainen et al. 2002).

Other quality attributes such as testability, understandability, and modifiability can be also implemented in the architectural design. This is achieved, for example, by specifying required levels of cohesion and coupling for the design (Davis 1990). Some of these quality attributes, such as safety, may need specific verification processes to ensure that these characteristics are included in the design (Rodríguez-Dapena et al 2001). Ebert (1997) suggests to use checklists-based inspections for measuring the success of quality attributes absorption into the architectural design.

An important purpose of the software architecture is to abstract the software design so that architects can analyse trade-offs between different solutions. Trade-off analysis means exploring the impact of certain decisions in terms of all the criteria and constraints (Ruhe et al. 2003). Based on a trade-off analysis, the architect must decide which requirements gets higher priority, and design a solution that adequately satisfies the requirements. A requirement might be sacrificed or compromised as a result of the trade-off analysis. (Hofmeister et al. 2000.) Sample approaches for analysing architectural designs against requirements include Architecture Trade-off Analysis Method (ATAM), Scenario-Based Architecture Reengineering (SBAR), and Architecture Quality Analysis (AQA) (MOOSE 2002a). In addition to trade-off analyses, these

methods can be used for evaluating absorption of quality attributes into the design.

As architects examine the requirements and develop possible architectural solutions, they might need to consult stakeholders to negotiate on changes to the requirements (Hofmeister et al. 2000). Ideally, software architects are involved in the early activities of the development. This way development of unachievable requirements is more likely to be prevented. Software's architecture must be considered during the development of requirements to make certain that the requirements are valid, consistent, and feasible. Architecture modelling and trade-off analysis are important activities and should already take place during requirements engineering. (Mead 1994.)

Software projects often consider requirements analysis and candidate architectural solutions concurrently. However, the relationship between requirements and software architecture is not obvious. Requirements describe aspects of the problem to be solved and constraints on the solution. In contrast, architecture describes a solution to the problem stated in the requirements. Requirements deal with such concepts as goals, options, agreements, issues, conditions, and system features and properties. They may be simple or complex, precise or ambiguous, stated in natural language or precise formalism. In contrast, the concepts used in architectural design differ from those used for requirements. Architecture deals with components, connectors, topologies, and the software's desired properties. These differences between requirements and architectures make it difficult to build a bridge that spans the two. (Grünbacher et al. 2001.)

Because of the different concepts used in requirements and software architectures, consistency and traceability maintenance between them is complicated. A single requirement may concern multiple architectural solutions and a single architectural element may have numerous relations to different requirements (Grünbacher et al. 2001). Nevertheless, software development standards require that traceability between an architectural design and requirements is established. Furthermore, the architecture's consistency with the requirements must be ensured. (ISO/IEC 12207; ISO/IEC TR 15504-5.)

3.3 Requirements and detailed software design

Detailed software design represents the software at a level of abstraction that is very close to code. The design must describe data structures, interfaces, and algorithms in sufficient detail to guide developers in construction of the software code. These descriptions are represented using design notations such as flowcharts, N-S charts, decision tables, program design languages (PDL), dataflow diagrams, entity-relationship diagrams and UML. (Pressman 2000.) Object-oriented design of embedded real-time software considers detailed design as definition of data structures, associations, operations, algorithms and exceptions (Douglass 2000).

Software designers should be included in the development of requirements (e.g. in requirements inspections) to make sure that requirements can serve as a basis for detailed designs. Several different software designs will usually satisfy the given requirements. These designs may vary in performance, efficiency, and robustness characteristics. The most efficient way to fulfil the requirements should be explored. (Wiegers 1999.)

Both functional and non-functional requirements affect the detailed design activity. While functional requirements steer the design's operations and algorithms, non-functional requirements set constraints on these solutions. For example, performance requirements oblige designers to develop algorithmic solutions that meet the given timing constraints (Davis 1990). Also hardware constraints, such as available memory, affect designers' work by restricting the solution space. Design constraints stated in software requirements affect the detailed design activity explicitly. These constraints may define the notations and techniques used in the detailed design.

Quality attributes appear in design trade-offs when designers need to choose between particular structural and behavioural aspects of the system. For example, quality attributes can be used to guide the selection and application of design patterns during design. Characterising and using patterns according to quality attributes allows designers to meet system-wide non-functional requirements. (Gross & Yu 2001.) As in architectural design, some quality attributes may need specific verification processes to ensure that these characteristics are reflected in the detailed design (Rodríguez-Dapena et al 2001).

Software development standards require that a software component's detailed design is traceable to software requirements and consistent with them (ISO/IEC 12207:1995; ISO/IEC TR 15504-5:1998). The design should be also verified so that all of the functional requirements are accommodated and unnecessary functionality is not included (Wiegiers 1999).

3.4 Requirements and software coding

The purpose of the software coding activity is to produce software units that reflect software designs and requirements. In coding activity, the developers begin to write modules that compile in the programming language (Mazza et al. 1996). If designs are specified detailed enough, they may be used directly as the basis for coding. Otherwise, developers need to use software requirements stated in the SRS and detailed designs together.

Implementation of functional requirements in code is more straightforward than is the case with non-functional requirements. Software engineers have to consider various aspects while implementing non-functional requirements. Performance requirements might oblige software engineers to use a low-level programming language (Ronkainen et al. 2002). Efficiency requirements, such as memory constraints, require programmers to use available hardware resources narrowly (Davis 1990).

Maintainability and portability requirements are also clearly present in the coding activity. The source code's maintainability is achieved by following the organisation's coding standards. Static code analysis tools such as QAC/++ (2003) or Logiscope (2003) can be used to identify a source code that is not conformant to the specified quality attributes. These tools typically identify problem areas in the code's structure, maintenance, and portability issues. Programming language selection, for example between a standardised high-level and a processor specific low-level language, also has considerable influence on portability requirements.

Certain quality attributes, for example predictability, should become crucial for developers at the coding activity. Definition of these characteristics cannot be made precisely at the beginning of the development because of their dependency

on the software architecture and implementation technology. (Rodríguez-Dapena et al. 2001.) On the other hand, trade-off analyses between some quality attributes have to be made before coding. For example, fulfilling an efficiency requirement may require that a code is implemented using a specific compiler and operating system. As a result, the system may work fast, but at the same time it will be hard to maintain, enhance, and port to another environment. (Wieggers 1999.) Some of these quality attributes may need specific verification procedures to ensure that the desired quality is elaborated into software code (Rodríguez-Dapena et al. 2001).

In some phase of the construction process, developers will encounter ambiguity and confusion while translating software requirements into code. To solve these problems, the obscure requirements should be traced back to their sources and a solution would be negotiated. Procedures should be defined for a situation where problems can't be resolved immediately. (Wieggers 1999.) Management of the confusion and ambiguity is improved when requirements are traced into code and their consistency is checked. Maintaining a code's traceability and consistency with requirements is also required by software development standards (ISO/IEC 12207:1995; ISO/IEC 15504:1998).

3.5 Requirements and software testing

The purpose of the software testing activity is to ensure that the produced software will satisfy the software requirements. Software testing includes three basic testing stages: module, integration, and system testing. Module testing verifies that a software module's source code is doing what it is supposed to do. Testing of software modules is followed by integration testing, where the focus is on software designs and the structure of the software architecture. Finally, system testing is performed to ensure that the software as an entity fulfils all software requirements. (Mazza et al. 1996; Davis 1988; Pressman 2000.)

System testing is the most significant testing activity from software requirements viewpoint. Davis (1988) emphasises that system testing should ensure that the entire software embedded in its actual hardware environment behaves according to the SRS. System test plans and test cases should be designed to ensure that all the software's desired characteristics are achieved. The purpose of test planning

is to assess how the software will be tested for conformity with the SRS. Another purpose of the planning is to ensure that the SRS is verifiable. An SRS is verifiable only when all of the stated requirements are verifiable. However, some of the requirements cannot always be rigorously verified. For example, usability requirements have multiple interpretations and are hard to measure quantitatively. (Davis 1990.)

Software testing has a major role in embedded system development due to the system's reliability and other non-functional requirements. For example, a system's quality attributes such as safety and reliability often require high code coverage percentages. Usability aspects could be tested by gathering suitable information during usability tests in an environment as close to the operational environment as possible (Ebert 1997). Other sample techniques for non-functional requirements testing include recovery testing, security testing, stress testing, and performance testing (Pressman 2000). However, the realisation of quality attributes is often difficult to verify. Various possible scenarios and different input combinations force developers to limit the set of test cases. In reality, quality attributes are often the first ones to bypass systematic verification. (Rodríguez-Dapena et al. 2001.)

The software testing activity's test cases are based on software requirements. The relation between requirements and test cases is considered so essential that the link between them should be always established (Weber & Weisbrod 2002). Every requirement should be traced to at least one test case, so that all expected system behaviour is verified (Wiegiers 1999). Composing test cases as soon as requirements stabilise enables developers to find and correct flaws in requirements inexpensively. Writing test cases for requirements crystallises the developers' vision of how the system should behave. Ambiguous and conflicting requirements should be revealed when the developer cannot describe the system's expected response to a test case. (Wiegiers 1999.)

In actual practice, software developers often fail to manage requirements, validation criteria, and test cases. Although it is agreed that requirements specifications should go hand in hand with validation criteria and test cases, the actual specification documents do not really fulfil these agreements. One reason for this is the fact that it is very difficult to specify validation criteria and suitable test cases for some functional requirements. With non-functional requirements the

case is even harder. Providing concrete requirements validation criteria examples for developers and having developers exploit and re-use these samples may improve these problems. The validation of non-functional requirements is also improved if these requirements are stepwise refined until they are implemented by a set of functional requirements. (Weber & Weisbrod 2002.)

4. Requirements implementation-supporting elements

The previous chapter discussed the role of requirements in software development activities and overviewed elements needed for effective requirements implementation in these activities. The importance of maintaining consistency and traceability between requirements and software work products was emphasised in all discussed activities. In addition, requirements change management is needed for controlling the inevitable change.

This chapter discusses these requirements implementation-supporting elements, which manifest themselves throughout the software development. First, requirements change management and its effect on software development is discussed. Then, traceability of requirements, which is also an important part of the change management, is covered in more detail. Finally, approaches for ensuring consistency between requirements and different software work products are presented. The presented requirements implementation-supporting elements are later used in Chapter 5 to develop the requirements implementation framework.

4.1 Requirements change management

Changing requirements are the reality in system development rather than stable ones. Designers are faced with unpredicted and disruptive changes in requirements that the system has to satisfy. (Harker et al. 1993.) Requirements sometimes change because of an error in requirements analysis, but more often it is consequence of a change in the system's environment (Sawyer & Kotonya 2001). In addition to the change in the system's operating environment, the hardware where the software is embedded may change. Therefore, concurrent development of the embedded system's hardware and software is a possible source for change. (Mäkäräinen 2000.) Sometimes, unfeasibility of requirements is only revealed once the actual software designs and code are implemented and tested; therefore, original requirements may need to be changed because of a change request from software development activities.

Whatever the reason for a change, it is important to manage the change by ensuring that the proposed changes go through appropriate review and approval procedures (Sawyer & Kotonya 2001). Effective system development requires organisational change management policies, which define the processes used for change management and the related information that is associated with change requests (Kotonya & Sommerville 1998). The need for requirements change management is also stated in the Software Engineering Institute's process improvement framework Capability Maturity Model Integration (CMMI). The requirements management process area in the CMMI framework requires that changes to the requirements are managed during the project (CMMI-SW 2002).

The need for a formal requirements change process arises when many changes to requirements are proposed. A requirements change management process consists of activities for documenting, reporting, analysing, costing, and implementing changes to requirements. Change management process can be thought of as a three-stage process. First, the problematic requirement is identified and analysed using the problem information and change for the requirement is proposed. Then, the proposed change is analysed to see how many other requirements are affected and what is the cost of the change. Finally, the change is implemented and relevant documents are updated. (Kotonya & Sommerville 1998.)

Procedures for requirement problem analysis and change implementation are dependent on the type of the change, the requirement, and requirements documentation. Cost and change impact analysis, however, are more general processes. These processes ensure that unnecessary changes are not made and that affected requirements are discovered. (Kotonya & Sommerville 1998.)

A requirements change management process may be included in a process that is intended for general change management in the system development. Examples of change management models include Olsen's change management model, the spiral-like change management model, and the generic change management process model (Parviainen et al. 2003).

A recent survey in embedded software industry revealed that there are problems implementing requirements change management process. Although change management procedures are defined, they are in many cases rejected. The main reason for this is that these procedures take too much time. Time is consumed

for example in re-reviewing all the relevant documents after every change. As a result of inappropriate change management, requirements and design documents may easily lose their consistency. (MOOSE 2002b.)

Change management process is also complicated by communication problems specific to embedded systems development. Concurrent development of an embedded system is often carried out in separate technology groups. Therefore, detailing the changes to all concerned parties becomes crucial when developers share the same software components. However, the development groups usually operate independently and their working practices, tools, and vocabularies are not always similar. (Mäkäräinen 2000.)

To overcome the problems in implementing a change, a change control board or (CCB) may be needed for controlling the change process. Such board is a body of people (possibly one individual or a group of people) that makes decisions about approving proposed requirement changes. Generally, a CCB reviews and approves changes to any baselined work product on the project. Large projects often have many levels of control boards. For example, some control boards are responsible for business decisions while others deal with technical issues. Smaller projects may have only one or two people that make all the change decisions. However, a CCB should have representatives from all groups that are affected by a change. Sample representatives include people from product management, project management, configuration management, software development, and quality assurance. (Wiegers 1999.)

4.1.1 Impact analysis

The impact of the change must be analysed before a requirement change is approved. Not only must the change's impact on the requirement specification be analysed; its impact on lower level work products has to be examined as well. For example, if a change is made to a single requirement that affects few details in high-level designs, it may affect many other lower-level designs, implementations and test cases. This kind of phenomenon is known as the ripple effect. The purpose of impact analysis is to tackle problems such as the ripple effect by identifying potential consequences of a change. Impact analysis can

also be used to determine what to modify in order to accomplish a change. (Arnold & Bohner 1993.)

Research on impact analysis has concentrated mainly on determining source code level changes or using traceability to predict changes to different work products created during the development. Techniques for impact analysis on implementation level include control and data dependency, and program slicing. Various traceability approaches enable impact analysis throughout the software development. (O'Neal & Carver 2001.) Wiegiers (1999) suggests that developers use checklists and defined procedures for discovering possible implications of a change. As a result of impact analysis, a developer reports to the CCB the needed effort for a requirement change. A standard reporting template will make it easier for the CCB to find information needed to make decisions.

Quality impact analysis methods such as Quality Factor Deployment can be utilised to assess the impact that a requirement change is likely to have on quality aspects such as performance, safety, and reliability. Also requirements inspections, viewpoint analysis and trade-off analysis can be used to identify potential conflicts and trade-offs within the set of changing requirements. (Lam & Shankararaman 1999.)

In actual practice, impact analysis is mostly performed manually, because tool support is weak. Also absence of traceability information between requirements and other software work products complicates impact analysis. (MOOSE 2002b.) Wiegiers (1999) states that the organisation's ability to succeed in impact analysis depends on the quality and completeness of the maintained traceability data.

4.1.2 Change management tools

When many requirements changes are introduced, the need to use a change management tool grows. Requirements change management may be supported by specialised requirements management and software configuration management tools. Some of these tools enable the use of database supported electronic change request forms. Furthermore, they may provide automatic notification to responsible persons with electronic forms. (Kotonya & Sommerville 1998.) Sample commercial requirements management tools that

also support requirements change management include Rational RequisitePro (2003) and DOORS (2003).

The general problem with change management tools is their own implicit model of change process, which organisations must adopt. In addition, special-purpose change support tools are fairly expensive and difficult to integrate. Therefore, these tools are mostly used by very large organisations in very large projects. Because requirements management also involves release management and configuration management, a tool support that integrates these features is actually demanded by industry. Some of the requirements management tools provide only some of the demanded features, while integration of configuration management, change management, and test case management is needed. (MOOSE 2002b.)

Requirements change management and software configuration management (SCM) have similar objectives. SCM is used as a means of controlling change throughout the software development process. It consists of activities that identify unstable work products, establish relations among them, define mechanisms for managing versions of these work products, control changes that are imposed, and audit and report about the changes that are made. Specifically, change control activity of SCM is responsible for ensuring quality and consistency as changes are made to configuration items. (Pressman 2000.) Requirements can be regarded as software configuration items and therefore controlled under SCM. The same SCM processes and tools that are used for design, implementation, and testing activities can be also used for versioning and managing changes in the requirements. (Crnkovic et al. 1999.) However, Kandt (2002) states that most configuration management tools are file based and therefore do not manage documents at a level of granularity that is truly useful - individual requirements, design concepts, classes, functions, test cases, and so on.

4.2 Requirements tracing

Requirements, designs, and implementations that are correct, consistent, and error-free ensure that the developed system meets its stakeholders needs. One of the key factors in achieving these features is understanding and tracing the relationships amongst these work products. (Palmer 1997.) The development of

embedded systems also requires that traceability between a system's different technology components is handled (Mäkäräinen 2000). Requirements traceability is defined as "...the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases)." (Gotel & Finkelstein 1994, p. 97)

Requirements traceability has various benefits in software development when used appropriately. As discussed earlier, it provides a means of managing change in requirements. Traceability can be also used for showing a system's compliance with requirements, maintaining a system's design rationale, proofing when a system is complete, and establishing maintenance mechanisms (Ramesh et al. 1995). Traceability is considered a key issue in transforming requirements into architecture and further work products (Grünbacher 2001). Design decisions are consistent with decisions made earlier in system development when lower level work products are traced to higher-level work products. Requirements tracing enables conflict detection by discovering linkage between selected entities and by providing visibility into the entire system. It also supports assessment of high-level behaviour and non-functional requirements' impact on design specifications. Also test cases coverage in relation to requirements is provided. (Palmer 1997.) Tracing requirements to designs, software units and all levels of testing has been also stated to be the best way to monitor progress (Lindström 1993).

A defined traceability policy is needed as the number of requirements grows. All decisions about which elements are to be traced and how to do so should be defined in the project's inspection phase (Weber & Weisbrod 2002). Factors that influence such decision-making are the number of requirements, the system lifetime, organisational maturity, the development team size, type of the developed system, and specific customer requirements. Small teams can handle changing requirements without structured traceability information, but larger teams have to establish a formal traceability policy. Certainly, the development of critical systems needs a more comprehensive traceability policy than the development of non-critical systems. (Kotonya & Sommerville 1998.)

Although requirements tracing is recognised as a very useful practice in software system development, it is often misused and poorly performed (Gotel & Finkelstein 1994; Lindström 1993; MOOSE 2002b). One of the main problems with tracing is related to the developers' necessity to manually add trace elements to requirements documents and following work products. Since these products have little or no direct consequence on the work of the development team, setting trace elements has a low priority. The benefits of traceability are not understood until later activities of the development life cycle, such as validation testing and system installation and operation. (Palmer 1997.) Another reason for problems with tracing requirements is that relations amongst documents (e.g. between requirements and architecture) are too complex to specify manually especially when projects are large (MOOSE 2002b).

Development environments for embedded software rarely provide any support for managing traceability issues (Mäkäräinen 2000). Common methods for requirements tracing include traceability matrices in the form of lists and tables. These matrices show relations among the requirements and relations between requirements and development elements (e.g., design, code, or test case). However, such methods require manual work and therefore have been regarded as laborious. Especially, manual maintenance of traceability links has been found to be difficult and error-prone (Mäkäräinen 2000). Tracing of quality attributes is considered to be particularly complicated (Ebert 1997).

Tools for automating the process of tracing have been presented, but the problem still remains in industry (MOOSE 2002b). Sample commercial requirements tracing tools include RTM (2003) and DOORS (2003). Introducing a requirements tracing tool is a big step for the company and often the tools do not satisfy the needs of the company nor the traceability. Reasons for rejecting the use of traceability tools are various. Requirements tracing tools have for example been criticised for ignoring the meaning of relations between requirements. The link between requirements is established but the rationale behind the link is missing. (MOOSE 2002b.)

Although requirements traceability is regarded as problematic to implement in practice, software development organisations should strive to utilise it. Software standards require that software work products are traceable throughout the development chain (ISO/IEC 12207:1995; ISO/IEC 15504:1998). The software

process improvement framework CMMI also emphasises the importance of traceability. Requirements management process area in CMMI requires that bi-directional traceability among the requirements and work products is maintained (CMMI-SW 2002). However, tracing requires a great deal of resources and therefore it should be a compromise reflecting costs and benefits of linkages. Tracing should be executed only where the traceability information is truly useful. (Stevens et al. 1998.)

4.3 Consistency management

Software work products at different levels of software development (e.g. requirements, designs, code, and test cases) are closely inter-related. Maintaining consistency among these work products is part of verification and validation processes in software development. Consistency among the work products ensures that the produced software will meet its specified requirements. Because requirements serve as a starting point for software development, consistency between requirements and following work products should be ensured throughout development. Also software standards (ISO/IEC 12207:1995; ISO/IEC 15504) and industrial experiences (Lindström 1993; Olsson & Runeson 2002) indicate that consistency between requirements and software work products has to be maintained at all levels of development. The software process improvement framework CMMI also states the need to handle inconsistency. The framework's requirements management process area requires the handling of inconsistencies between requirements and work products (CMMI-SW 2002).

Sources for inconsistencies between different software work products are various. Software engineering is conducted by human beings and therefore errors may be introduced in any development activity by the developers of the system. Inconsistencies may also arise when an organisation's personnel and customers change. As analysts and customers leave a development project, they are replaced by others who might drive development in a different direction. As a result, inconsistencies between new requirements and existing software work products arise. (Robinson & Pawlowski 1999.) Other possible sources for inconsistency are approaches and tools used in the software development. The problem with current development approaches is that relating information across work products is either not straightforward or not possible at all. In addition, most of the development

tools support only specific development activity with limited support to other activities in the development process. (Olsson & Grundy 2002.)

Lindström (1993) reported typical problems in software development when requirements consistency with lower work products is not closely monitored. First, requirements were analysed and high-level designs based on these requirements were derived at the beginning of the project. Then, detailed designs were developed based on the high-level designs with little attention to the source requirements. Similarly, source code and unit tests were derived from the detailed designs, which removed the development further from the requirements. Not until integration and testing activities did the focus return to the source requirements. By this time, some requirements had changed, new requirements had been added, and some requirements had been lost or forgotten. The result was that inconsistencies between requirements and other software work products produced an incorrect product. The lesson learned from this project was that requirements compliance has to be somehow measured at each step of the development. To achieve this, Lindström suggests tracing of requirements and incorporating a requirements check in work product reviews at all levels of development.

As the size and complexity of the software continues to grow during the development, software developers are faced with problems in preserving consistency between different work products. These problems are tackled with a consistency management process, which is composed of various activities such as the detection of overlaps, consistency checking, diagnosis, consistency handling, consistency tracking, and specification and application of consistency management policy (Kozlenkov & Zisman 2002).

Ensuring consistency between different work products and requirements has been a research subject for some time now. The researchers have, for example, proposed using inspections, quality models, and formal methods on proofing consistency between requirements and following work products. Some of these methods for design, coding, and testing activities are presented next.

Design

Despite the close relationship between requirements and software designs, little attention has been paid to their integration. This has increased the risk of inconsistencies in software development. A sample approach to reduce the risk of inconsistency between requirements and architecture has been proposed by Inverardi et al. (2001). They propose an approach that traces co-ordination requirements from their definition to the low-level specification. The architecture is then validated with respect to these co-ordination requirements. Other sample methods to strengthen the relationship between requirements and architectural design include CBSP (Component-Bus-System-Property) method (Grünbacher et al. 2001) and the combined use of goal-oriented language GRL and a scenarios-oriented architectural notation UCM (Liu & Yu 2001). Chechik and Gannon (2001) present an approach where lightweight formal techniques are used for automatic analysis of consistency between software requirements and detailed design. Design inspections can also serve as a consistency checking approach, as is shown by Travassos et al. (1999).

Coding

Approaches for ensuring consistency between requirements and software code have also been presented. Robinson (2002) suggests an implementation approach where an instrumented code is used to ensure that requirements are traceable and satisfied. Chechik and Gannon (1995) propose a tool called Analyzer which discovers instances of inconsistencies and incompleteness in implementations by proofing state transitions. Punter et al. (2002) present a method for evaluation and correct implementation of non-functional requirements in software designs and code. They suggest combining the use of quality modelling, probability concepts, and measurement techniques in order to evaluate code and design artefacts. Phased inspection technique according to Knight and Myers (1993) can be used for checking the existence of appropriate functionality and quality characteristics such as portability, reusability, and maintainability in code.

Testing

Testing can serve as a consistency checking approach throughout embedded system development. Co-design of hardware/software systems relies heavily on testing that is executed at all levels of design activities. When a consistent set of

test cases is used, consistency among different design levels can be ensured. To support this, Gupta et al. (2001) propose an automatic test scenario generation approach. Their method is based on a state-based model of the proposed system. Another sample requirement-based approach for rigor verification of software systems is suggested by Tahat et al. (2001), where Specification and Description Language (SDL) is applied to create finite state machines, which are used to automatically generate test cases that are consistent with the requirements.

5. Requirements implementation framework

In order to analyse and improve an organisation's capability to implement requirements effectively, a framework is needed to describe how requirements are ideally implemented throughout the development. This chapter presents such a framework, which integrates elements of requirements implementation referenced to previously in Chapters 3 and 4. These elements were found in the literature as good practice of requirements implementation. While the previous chapters provided examples of how these elements could actually be implemented (summary available in Appendix 2), this chapter discusses their interrelations and integration.

5.1 Requirements implementation framework for embedded software

The purpose of the proposed requirements implementation framework is to describe what kind of elements are needed for effective implementation of requirements during embedded software development. These elements are, for example, requirements implementation processes, methods, and roles. Because elements such as a change control board, traceability, and extensive reviews require considerable resources, smaller and more flexible organisations may need to tailor the framework's concepts. Tailoring of the framework is discussed at the end of this chapter.

The software development V-model presented in Chapter 2 is used as a basis for the framework for its clarity and simplicity in presenting related software development activities, not because it would describe how software is actually developed. The activities shown in this model are generally acknowledged in software development, but their interrelations and performance order may vary.

The requirements implementation framework is presented first at a detailed level from the viewpoints of requirements analysis, designing, coding, and testing. Requirements implementation elements in these activities are classified to precondition, process, change management, and verification and validation sections. The precondition section includes elements that need to be in order before successful requirements implementation in the activity is possible. The

process section then presents elements that are utilised during the activity. While the change management section discusses the important elements related to requirements change, the verification and validation section concentrates on elements that ensure the correctness of the produced work product in the activity. After the individual activities with related requirements implementation elements have been discussed, they are integrated into one holistic view.

The notation used in depicting the framework is shown in Figure 5. Development activities, such as software requirements analysis, are shown as rectangles. Work products from these activities are depicted as rounded rectangles. Human figures describe roles, such as a software designer or a CCB. Relations among activities, work products, and roles are shown as lines. A more specific description of a relation is given inside or at the end of the line. The relation can be either strong (solid line) or weak (dotted line). A strong relation implies that the relation is essential from the viewpoint of requirements implementation. A weak line indicates that the relation is sometimes questionable and, for example, dependent on the nature of the development project or product. Finally, a semi-dotted line compiles related elements together.

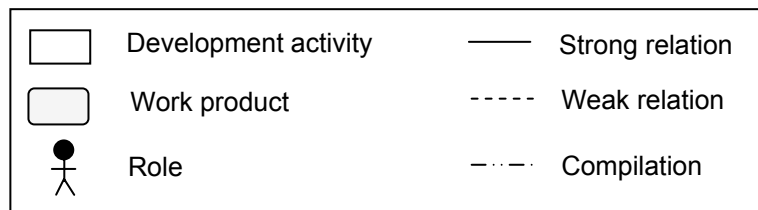


Figure 5. Framework notation.

5.1.1 Requirements implementation in software requirements analysis

Requirements implementation in software requirements analysis can be regarded as an activity which transforms high-level requirements into low-level software requirements and defines these requirements so that their further implementation in the following development activities is possible. Requirements implementation elements in software requirements analysis are shown in Figure 6. These

precondition, process, and verification and validation elements according to Section 3.1 and Chapter 4 are discussed next. Furthermore, change management elements according to Section 4.1 are covered below.

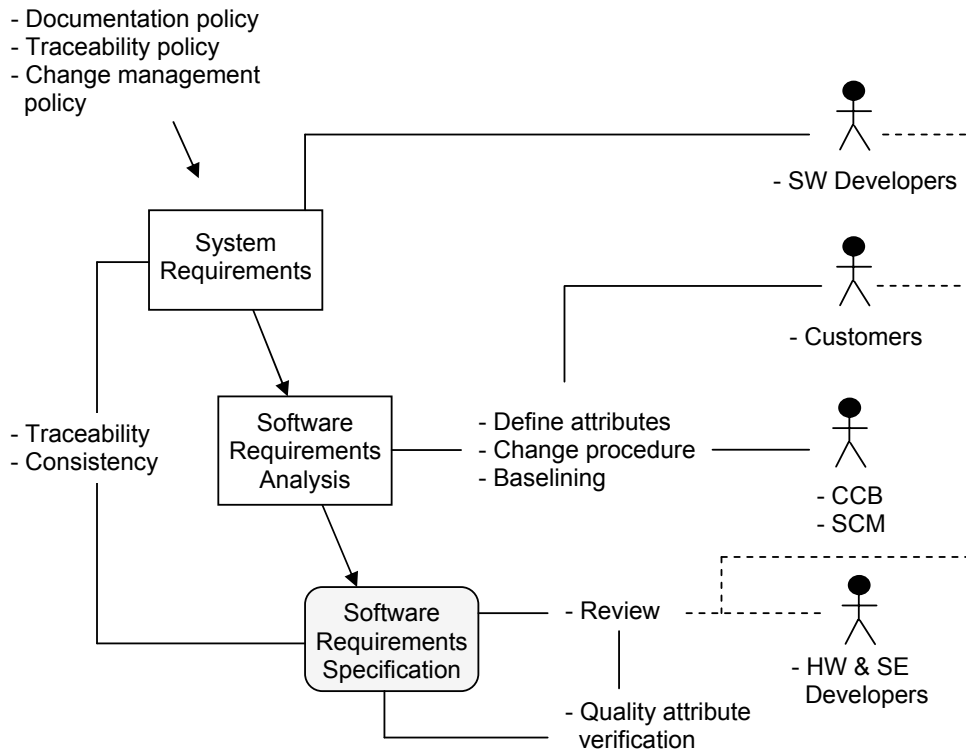


Figure 6. Requirements implementation in software requirements analysis.

Precondition

To increase the success rate of a development project, system level requirements development and allocation should include people with software expertise. These people may be software analysts, designers, and testers. Software developers' knowledge of the software's capability and cost-effectiveness to implement a certain requirements may affect the decision to allocate the requirement into either software or hardware requirement.

Before the implementation of system requirements into software requirements can begin, certain working policies must be defined. The most crucial are a

requirements documentation policy, a change management policy, and a traceability policy. These requirements policies need to be agreed to make sure that requirements are later handled uniformly and correctly. For example, a requirements documentation policy may define how requirements are classified and identified in requirements documents. A traceability policy is needed to agree and inform developers about which work products need to be traced and how tracing is performed. A balance between the cost and the benefit of having traceability information among work products should be determined. A change management policy defines responsible roles and change procedures in development activities when a change is introduced. Leaving requirements policies undefined inevitably creates later a situation where working methods are forgotten and not performed properly.

The above-mentioned requirements policies may be organisational-wide or specific to a certain project. They may be even stated before system requirements elicitation begins. Nevertheless, the definition of requirements policies must take place before software development activities begin. It is also important to inform the relevant developers about these policies. Developers must understand the benefits and rationale behind certain working methods. Only in this way will the developers be motivated to follow the defined requirements policies.

Process

During the software requirements analysis, certain attributes for requirements must be defined to enhance requirements implementation later. Requirements need to have information about their identity, priority and stability. Identification is needed for various elements of requirements implementation, such as traceability, consistency check, and change management. Priority information is needed to make design decisions. Valuable sources for requirements priority information include customers and software developers. Requirements stability information helps to control change of requirements. Defining a requirement's stability may require software design and development expertise. In addition to these attributes, documentation of requirements' source, scope and rationale may be usable later.

Change Management

An unmanaged change in the requirements during software development may easily cause the system to be inconsistent. The requirements analysis activity must have a procedure for managing changes from system level and from lower level development activities. This procedure has to make sure that the software requirements are consistent with the system requirements and internal consistency among the software requirements is preserved. To support impact analysis of a change, traceability amongst the software requirements has to be maintained. Software developers also have to have procedures for reporting and negotiating with system engineering level about problems implementing certain requirements in lower-level software development activities.

Baselining, or freezing a set of software requirements, is needed because changing requirements cause too much instability in software development. Although elicited requirements are not necessarily the final ones, a baseline of requirements should be defined to enable consistent development of both hardware and software. To baseline requirements and manage requirements change, a change control board is needed. A CCB for embedded system development may include representatives from software, hardware, and system disciplines, and even people from hardware quality assurance and hardware configuration management. For smaller and experimental projects, the CCB may be very flexible and only consist of a few people. Both change management and baselining are related to software configuration management, which controls the change throughout the software development.

Verification and validation

The internal and external validity of the SRS must be ensured in a review. Customers, software designers, and testers may take part in such review. Their views on the adequacy of certain requirements may prevent approval of incorrect requirements. Because the software is part of a larger embedded system, an important purpose of the SRS is to determine the interfaces from software to the system and hardware elements. Therefore, the presence of persons responsible for system and hardware elements may be also required in the review. Finally, the review of the SRS must include verification of quality attributes, which will ensure that non-functional requirements have been sufficiently acknowledged in the software requirements.

5.1.2 Requirements implementation in software designing

Software design activity elaborates the software requirements into designs, which can be used as blueprints to implement the desired software. Requirements implementation in software designing includes elements that allow developing correct and consistent designs against the requirements, and to analyse and implement requirements changes. Both software architectural design and detailed design activities include similar requirements implementation elements; therefore, they are both discussed in this section. Requirements implementation elements in software design are shown in *Figure 7*. The precondition, process, and verification and validation elements covered in Sections 3.2, 3.3, and Chapter 4 are discussed next. Also change management elements covered in Section 4.1 are discussed.

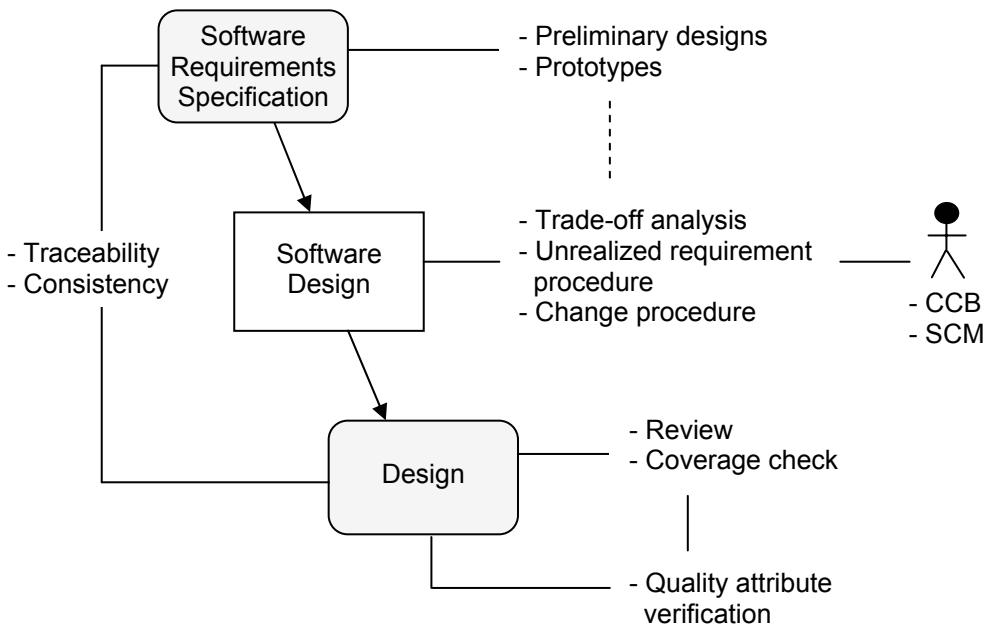


Figure 7. Requirements implementation in software architectural and detailed design.

Precondition

A good practice would be to include software designers in the specification of software requirements, or even during the development and allocation of system requirements. Their ability to evaluate and test proposed requirements in the early phases of system and software development may be invaluable. Demonstrating the system to the customers with preliminary designs and prototypes is a cost-effective way of eliciting new essential requirements and refining already agreed requirements. The results of preliminary designs and prototypes could be also used later in the actual design activity as a basis for design trade-off analysis.

Process

Design trade-off analysis ensures that requirements are implemented in the most effective way and resolves conflicts when requirements collide. To make decisions about which requirement is important to implement in the design, the designer needs prioritisation information of requirements. Other information that the designers may need from requirements include the rationale and source behind a requirement. These requirement attributes enable designers to justify design decisions, help to negotiate about fuzzy requirements, and make requirements reuse possible. Specifying software requirements precisely and quantitatively where possible during the requirements analysis activity pays off now when designers have to make design decisions. However, requirements should not be written as restricting design decisions, because they may prevent designers discovering the most effective solution to a requirement.

Change management

Designers should have change procedures for making modifications in their designs. These procedures may be jointly co-ordinated with the CCB and SCM. Change procedures have to include impact analysis of the change for software designs. Performing impact analysis produces much needed information for project scheduling and resourcing. Without impact analysis a change of requirements might result in a sequence of unsystematic changes to designs and overrun of the project schedule.

Besides the defined requirements change procedure, software developers need a procedure for unrealisable requirements. The developers will almost certainly

face ambiguity and conflicts in the requirements. Therefore, they need to report about the problematic requirement and ask for clarification. An unrealised requirement should finally be dealt by the CCB, which decides how the requirement is changed.

Verification and validation

When software designs are ready for implementation, they need to be reviewed for consistency with requirements. The review must also validate that all requirements are covered. Requirements need to be individually identified to enable consistency and coverage check. The same is expected in all other work product reviews during software development. For some quality attributes, a specific verification process is needed. This may be achieved, for example, by defining and using checklists for specific quality attributes in design reviews.

5.1.3 Requirements implementation in software coding

In the software coding activity, software requirements are finally implemented in a programming language that is understandable to the machine. Requirements implementation in this activity is considered an activity to ensure that requirements are properly realised. Preparing and analysing changes to software code when software requirements change is also an important element. Requirements implementation elements in software coding activity are shown in Figure 8. These requirements implementation process and verification and validation elements according to Section 3.4 and Chapter 4 are discussed next. Furthermore, change management elements according to Section 4.1 in software coding are covered below.

Process

The relationship between code and requirements is not as strong as is the case with higher level designs. This is true especially when detailed designs are very specific and enable automated code generation. On the other hand, if detailed designs were not properly done, software requirements may support the construction of code.

Change management

Developers that are responsible for software implementation must be ready for changes in the requirements. Procedures for change management need to be defined before the development begins. Change procedures should be planned and executed jointly with the CCB and SCM. Impact analysis is a significant part of the change procedure. Because of the close relation between implementation and detailed design, the impact analysis must consider both of them simultaneously. For example, a change in detailed design might require many changes in code, which in turn requires changes into detailed designs, etc. Developers also need procedures for unrealised requirements similarly to those in design activity. Some requirements' unfeasibility for implementation is not revealed until the actual code is written and tested.

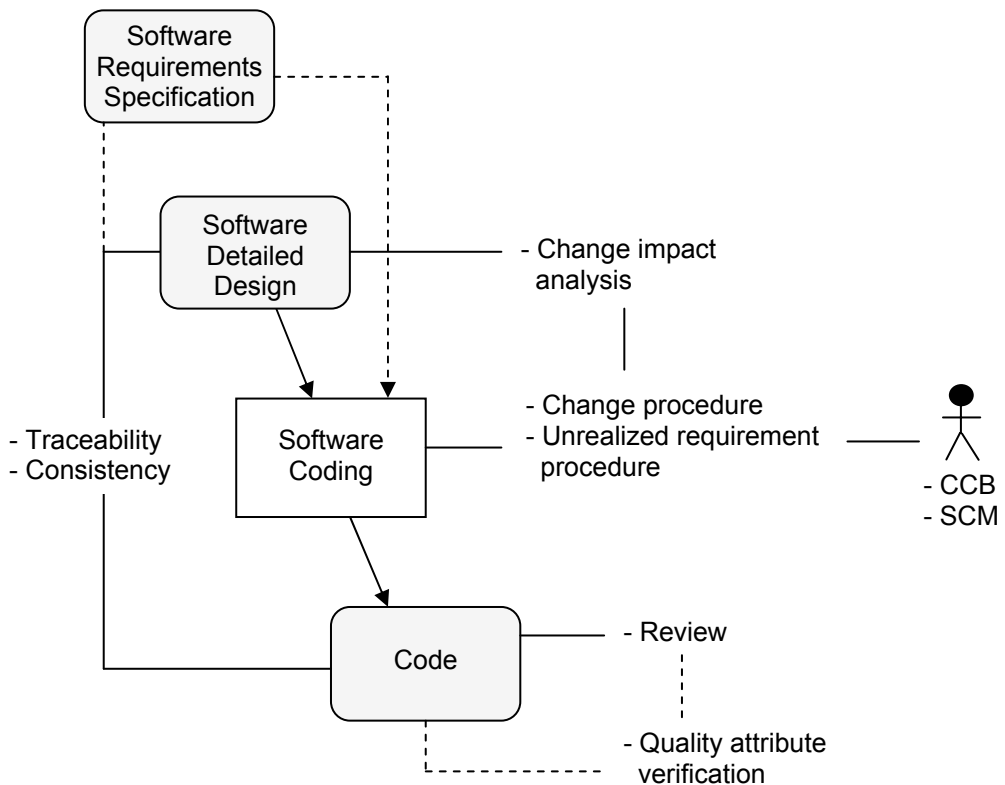


Figure 8. Requirements implementation in software coding.

Although software standards (e.g. ISO/IEC 12207 and ISO/IEC 15504) require traceability between software code and requirements, traceability links between software code and software requirements is not always a necessity. If the code is compatible with the detailed design, then all change management, consistency checking, and other procedures concerning the requirements are dealt between them. For example, the code may be automatically generated from detailed designs, and all the needed changes are made to these designs. Once the changes to the designs have been implemented, the code is regenerated. In this case, traceability between requirements and the code is replaced by traceability between detailed design and the code.

Verification and validation

Finally, when a functionality, a component, or some other part of the implementation is ready, it needs to be reviewed for correctness. If the software code is transformed from the detailed designs, its consistency with the designs must be ensured. Otherwise the code's consistency directly with the software requirements must be checked. A software code review process may also include a verification process for specific quality attributes.

5.1.4 Requirements implementation in software testing

Requirements implementation in testing refers to the elements that support testing the software rigorously against the requirements. Software testing consists of sequential activities such as module, integration, and system testing. Requirements implementation elements are mostly related to system testing. These elements and their relation to software testing are shown in Figure 9. The precondition, process, and verification and validation elements of requirements implementation covered in Section 3.5 and Chapter 4 are discussed next. Also change management elements covered in Section 4.1 are covered below.

Precondition

Requirements must be stated unambiguously and quantitatively to enable development of correct test cases. The SRS must have validation criteria for accepting certain requirements. These criteria should be defined as precisely as possible. Testers may already start planning test cases while software requirements

are being elaborated. Writing test cases for requirements during their development reveals possible flaws inexpensively.

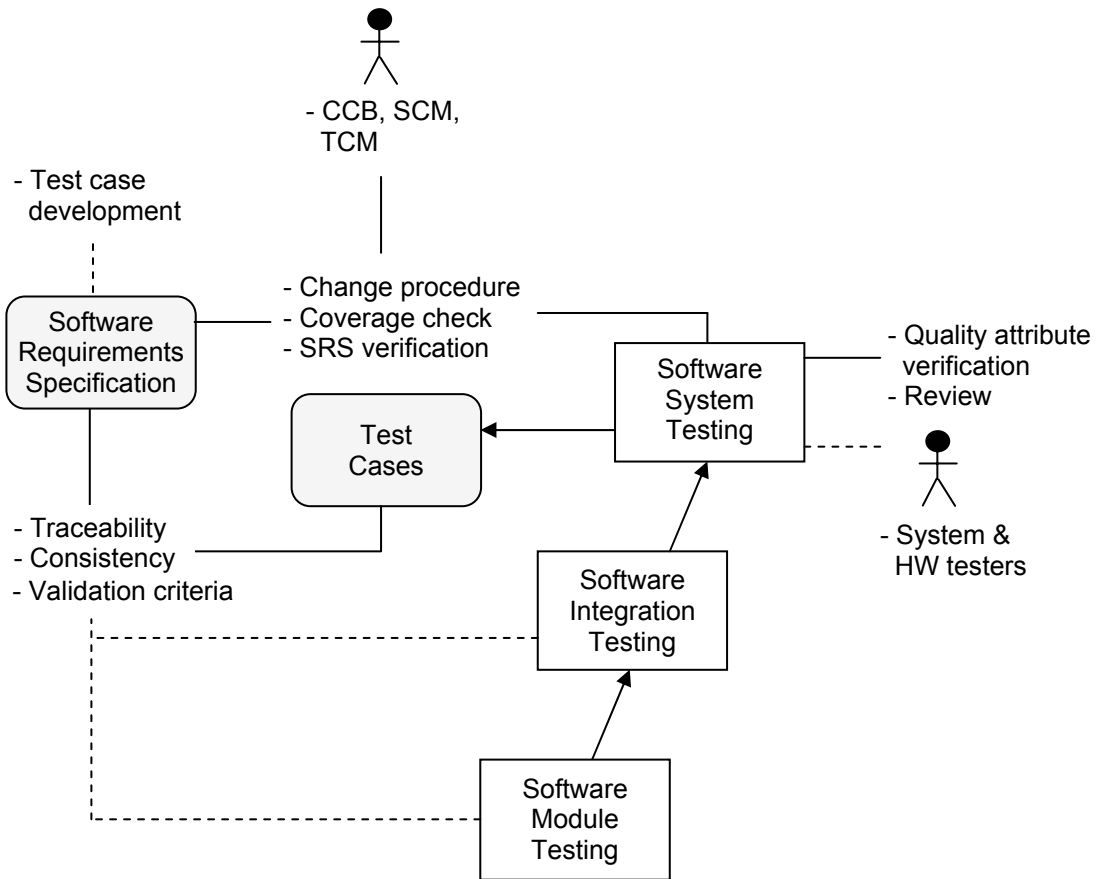


Figure 9. Requirements implementation in software testing.

Process

Module and integration tests ensure that requirements stated for architectural and detailed designs are correctly fulfilled. Module testing is often performed simultaneously with the coding activity. For some systems it may be necessary to trace these lower level tests to the requirements and check that all behaviour and quality attributes are covered. However, if software designs' traceability and

consistency with requirements are maintained, then there should not be any need for direct linkage between requirements and module and integration test cases.

The purpose of system testing is to ensure that the developed software works correctly in its actual environment and reflects correctly system and hardware requirements. Therefore, embedded software testers may need to have co-operation with system and hardware testers throughout software testing.

Change management

Since software testing is strongly related to requirements, a change of a requirement also affects testing activities. To maintain consistency throughout the development chain, testing activity needs procedures in case of a requirement change. For example, a change procedure may include an evaluation of what test cases in system testing need to be updated. Maintaining traceability between test cases and requirements facilitates this kind of evaluation. If module and integration tests are also dependent on requirements, then change procedures need to be defined for them as well. The CCB, SCM, and test case management (TCM) play a vital role in defining how the change is implemented.

Verification and validation

Checking consistency between requirements and test cases, for example in a review, will ensure that correct aspects are being tested. Testing of some quality attributes may require specific testing arrangements. For example, testing of usability may require usability tests that include users. Maintaining test cases traceability with requirements enables to check that all requirements are covered. Furthermore, satisfactory implementation of requirements can be proven to customers, when system test results are traceable to customer requirements.

5.1.5 Requirements implementation throughout the development

The previous sections discussed the elements of effective requirements implementation and interrelations of these elements in separate software development activities. In this section, a summary is made by combining the most essential elements into one holistic view. These elements should compose a

chain of supporting activities for software development that enable correct and effective implementation of requirements. Implementation of requirements throughout the development is shown in Figure 10.

Software developers should be considered before the actual software development begins. Their involvement during system requirements development may save the project from expensive changes in later development phases. Also documentation and working policies regarding requirements in software development need to be defined. Having clearly defined policies and informed and motivated developers on these policies will ensure successful handling of requirements.

One of the most important elements of successful requirements implementation is obviously tracing software requirements backwards to system requirements and forwards to software work products. Traceability is needed for requirements change management, consistency assurance, and coverage checks throughout the development. In the proposed framework, traceability covers all development activities except for coding, module, and integration testing. To introduce traceability into the software development successfully, it should elaborate seamlessly into software process through methods, techniques, and notations used in requirements analysis, design, code and test. Indeed, this requires a clear traceability policy, requirements identification and automation of traceability information maintenance.

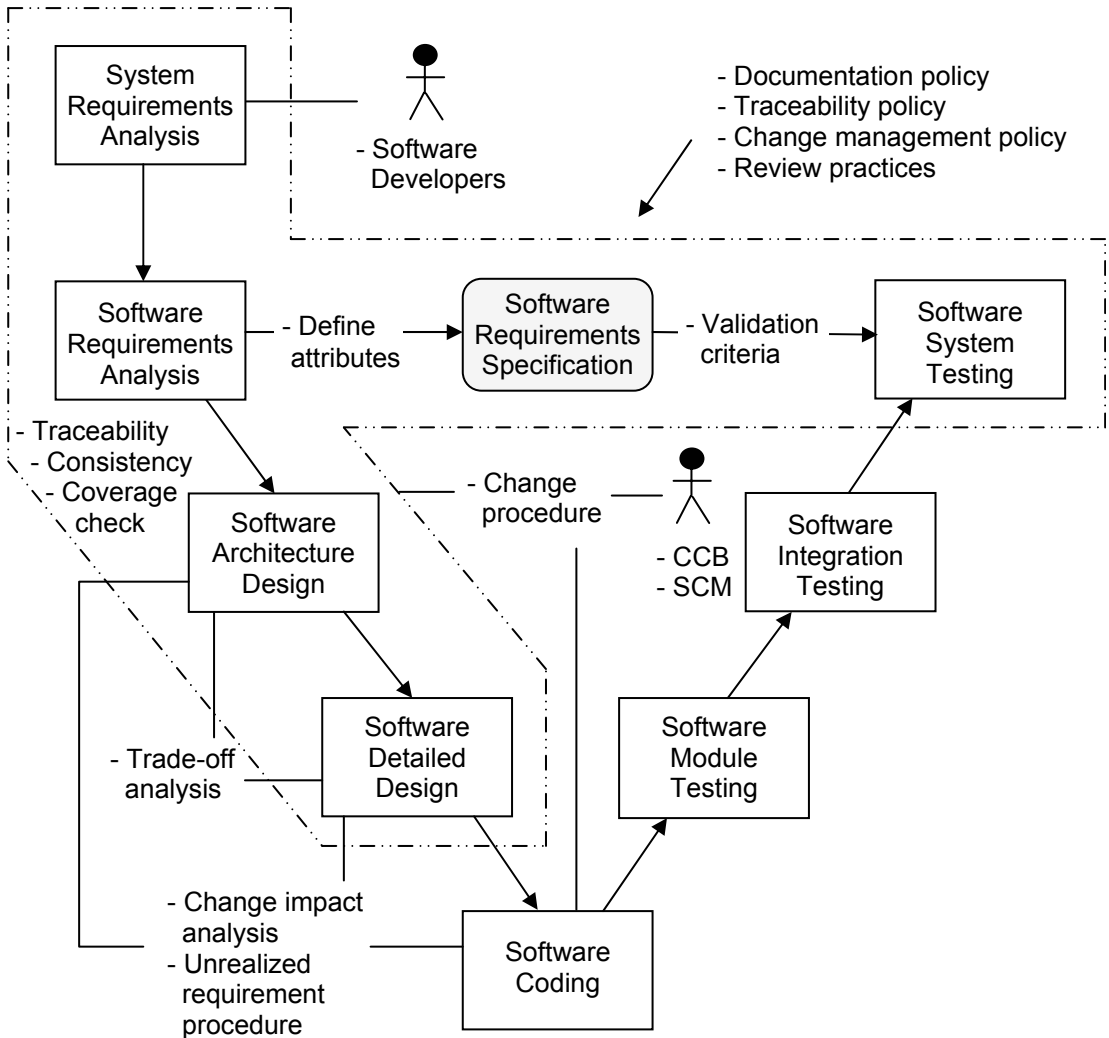


Figure 10. Requirements implementation throughout the development.

Requirements traceability is more challenging to implement in an iterative-like software development process than it is in a waterfall-like process. The waterfall model's sequential and clear order of activities helps to add traceability information to different baselines. However, the benefits of traceability are significant in iterative development where the change of designs, implementations, and test cases is continuous. Regardless of the development process model, ensuring requirements consistency and coverage during

development activities is facilitated, if the development organisation traces requirements to work products and has established review practices in use.

Another important element of requirements implementation is management of requirements change throughout the development. To avoid uncontrolled changes in the developed system and software, the organisation should have a CCB that manages changes. Because of the close relationship between requirements change management and SCM, their integration may bring valuable benefits. Requirements change management should include change management policy and change procedures for separate activities. Stability of requirements must be known to prepare designers for requirements change. In addition, impact analysis, in case of a change, needs to be done in design and coding activities. During these activities developers must also perform trade-off analysis and be ready for unrealisable requirements.

The proposed framework assumes that software code can be directly created based on detailed designs. Thus, traceability and consistency between the code and the requirements is maintained through detailed designs. The framework also assumes that module and integration tests are based on detailed and architectural designs, and that their mutual traceability and consistency is managed without any direct relationship with requirements. These and other assumptions in the framework may be different from an organisation's actual software development process. Therefore, possibilities to tailor the framework for different development projects and organisations are discussed in the next section.

5.2 Adaptation of the framework

Constructing a requirements implementation framework that is suitable for analysing and improving requirements implementation in different kinds of development environments, life cycles, and processes is not an easy task. Furthermore, to develop a framework that covers all the essential implementation elements, but is not at the same time overly complex, presented problems. The relationships and dependencies among software development activities and requirements implementation elements are clearly not straightforward.

The need and ability to sacrifice resources to requirements implementation is dependent on both the size of the developing company and the size of the project. More strict approach is suitable for large companies and large complex projects that have a number of teams in different disciplines developing the same product. In contrast, small companies and small projects need a more informal and flexible approach. Small and large companies' needs and resources are obviously not comparable. In a small organisation, for example, reporting an unfeasible requirement may be informal communication between a developer and a project manager. Nevertheless, roles and procedures regarding requirements implementation in both small and large organisations should be clearly defined.

The nature of developed embedded product also governs the attitude towards requirements implementation aspects. Product and also project requirements direct how the software development is carried out. For example, development of a safety critical system may require an extremely rigorous approach in the implementation of the given requirements. In contrast, an experimental project may have very flexible procedures for requirements implementation. The demands for the software process in the development of an aircraft are quite different from those in the development of an electronic alarm clock.

To enhance requirements implementation, the proposed framework's elements need to be adapted for the project's development life cycle. The elements and their interrelations can be tailored to serve as a good software practice for incremental, evolutionary, or any other development life cycle. For example, if the developed product requires a risk driven development approach such as spiral development model (Boehm 1988), the requirements implementation elements shown in the framework may be iterated with the development activities. The purpose of the framework is not to supplement the traditional sequential development life cycle's deficiencies on requirements implementation concepts. On the contrary, it should be tailored for the needs of any life cycle, project, and product.

Before requirements implementation in software development can be improved, the system requirements engineering processes must be in order. Stakeholders needs must be understood and conformed. This requires a firm grasp of the requirements elicitation and validation processes. Also traceability and other

requirements management elements must be in order when stakeholder needs are developed into system requirements and further allocated into software requirements.

Literature study revealed that developers' motivation to use some of the framework's elements during software development is hard to achieve. For example, elaborating effective traceability and consistency check practices into a software development process is regarded as time-consuming and expensive. Gaining developers' acceptance of these aspects would be very challenging. They are already over employed and certainly do not need more bureaucracy that does not explicitly help doing their work. By explaining the rationale behind requirements implementation elements for developers, the chance for their commitment to these is increased.

6. Validation of the requirements implementation framework

Two case studies were conducted to validate the requirements implementation framework's capability to analyse and improve requirements implementation practices in an industrial environment. This chapter presents these studies and discusses the validation of the framework. First, the research approach used in the validation is overviewed. Then, the two case studies are discussed in more detail and the most important findings are presented. Finally, the framework's applicability is evaluated based on the results from the case studies.

6.1 Research approach for validation

Software engineering as a science discipline should involve an experimental component to test or disprove suggested theories. Researchers cannot rely merely on conclusions following their logical thought. Indeed, experiments have to be made with models, methods, and techniques to find out how and when they really work, to realise their limits, and to improve them. (Basili 1996.)

Experimentation of the suggested model in this study was achieved with an historical research approach according to Zelkowitz and Wallace's (1998) categorisation of software engineering validation methods. Historical research approach allows researchers to study organisations' legacy data and lessons learned by exploring existing software artefacts and by interviewing personnel. This approach was used in this study to evaluate the current state of requirements implementation practices in the case organisations.

The sheer historical validation approach was further extended by suggesting improvements to case organisations' requirements implementation practices based on the framework and by having the organisations' personnel to evaluate these suggestions. The framework was found to be useful if it could be successfully used in discovering problems in case organisations' current requirements implementation practices and suggesting valuable improvements to these problems. This condition was evaluated by analysing case organisations' feedback on suggested improvement proposals.

Data for the current state analysis of requirements implementation practices in the two case organisations was collected from multiple sources. First, problem areas of requirements implementation in the organisations were surveyed with an open ended questionnaire based on the questions in Appendix 1. Then, the discovered problem areas were further examined with more detailed questionnaires, interviews, and by investigating organisations' software related documentation.

By comparing the framework's elements and problem areas in a case organisation's current practices, an improvement proposal was composed. If a case organisation's development process lacked an element from the framework, its negative consequences were examined in more detail and a possible solution was considered. All solutions were then combined into an improvement proposal. This included processes, methods, and tools that the researcher believed would enhance requirements implementation practices. The summarised list of requirements implementation practices and methods in Appendix 2 was used as the basis for concrete improvement suggestions.

The improvement proposal was then reviewed by case organisations to get feedback on improvement suggestions. This feedback was used as evidence for validation of the requirements implementation framework's applicability. The validation of the framework was augmented by having two different kinds of embedded software developing organisations as subjects. The first studied case organisation is a large company that develops various products and has hundreds of people developing software. The second organisation is considerably smaller with several dozen developers and is focused on developing few products.

6.2 Case one - large organisation

The first case organisation builds embedded systems for document processing. The organisation's R&D consists of over 2000 people, including some 700 software developers. Software development approaches include application, control, and embedded software. The software is developed with a tailored development model from the classical V-model. Currently, the software development is under a process improvement program and the goal is to introduce CMM level 2 key process areas.

Requirements are supplied by marketing, maintenance, and R&D departments. These requirements are elaborated with safety rules, legislation, and standards into more detailed product specification. At the product level, requirements are initially specified in natural language and then developed into use cases, and sequence diagrams. Product requirements are decomposed into sub-system requirements, which contain requirements for mechanical, electronic, and software disciplines. This decomposition can continue to more detailed sub-systems, until requirements are implemented in mono-disciplinary levels.

For this study, requirements implementation practices in a software development unit of the R&D were explored. The unit's software integrator acted as a source of information for the inquiries. Also documents such as software process descriptions, SRS templates, and inspection checklists were examined.

6.2.1 Current state analysis

Software level requirements documentation in the case organisation was well managed and performed. For example, the organisation had defined a requirements management policy and a template for requirements specifications. The SRS template, however, lacked individual requirements' attributes such as stability, rationale, source, and priority. But these insufficiencies had already been noticed and improvement actions had been initiated. Requirements management was also planned to be improved with a commercial tool.

Internal verification of SRS's and baselining the requirements were also well handled. Consistency checking between software requirements and other software work products was also conducted. All development activities included review practices, where consistency was ensured. Checklists and other guidelines for the reviews imposed to check internal and external consistency of the work products.

Another well-controlled aspect, quite surprisingly, was requirements change management. Although the organisation reported having changing requirements as quite common phenomenon, it had established such change management practices that these changes did not have a negative effect on development activities or

work products. The organisation had established a change management process with a change management tool and a change control board.

Clearly the most serious problems that hinder requirements implementation in the organisation were with system requirements development and management. Although software requirements were well documented and managed, system requirements tended to be documented insufficiently and managed poorly. There was no system level requirements management process nor was there a common requirements documentation policy. This made tracing of software requirements to system requirements difficult. Furthermore, insufficient participation of software engineers in system requirements development and allocation lead to problems later when software development begins.

Problems also seemed to arise because of inadequate co-operation among software, hardware, and system levels. For example, informal checking of software requirements consistency with system requirements had later caused problems in system integration and testing because software and system requirements were not consistent after all.

Although software requirements change management was well managed, impact analysis of a change sometimes caused problems. Impact of a change was estimated by team-leaders before approval in the CCB. Sometimes these estimates were too vague and caused problems for project schedules. Other minor problems related to requirements implementation included the software architecture's analysis against requirements, testing of quality attributes, and lack of traceability information from requirements to design and code.

6.2.2 Improvement proposal

In order to improve requirements implementation in software level, the organisation must pay attention to system requirements development, allocation, and management. System requirements development and allocation procedures must include software engineers who could use measurement information and prototypes to specify development cost of certain software solution. Documentation and management of system requirements must be also enhanced. Organisation wide requirements policy containing procedures and templates

concerning all disciplines would enhance development, communication, and reuse of requirements. A common policy would also allow the use of one requirements management tool for maintaining all requirements.

Involvement of other disciplines in verification of software requirements should be improved. Software requirements must be consistent and compatible with hardware right from the beginning. In the past, hardware engineers were not involved in SRS reviews at all. Fortunately, this situation is now changing and other disciplines are more involved in these reviews. The current practice of informal reviews may be evolved towards more formal inspections with reading techniques such as PBR to make the involvement even more effective.

Implementation of requirements in the software's architecture could also be improved. Software's architectural design is currently very much dependent on the experience of the software architects. The architects manage to get the system working, but whether it can be developed to be the most efficient solution is another issue. Therefore, the architecture's trade-off and quality analysis methods such as ATAM, SBAR, and AQA could be utilised.

A few improvements in the coding and testing activities could be also made. Firstly, static code analysers could be used to identify areas of the source code that are not conformant to certain quality attributes. Analysers such as Logiscope and QAC/++ would automatically discover problems in maintainability, readability, and portability issues. Secondly, the requirements specification's quality attributes could be quantified and verified already in the software level tests rather than postponing quality attribute testing to the system level.

Finally, traceability information from software requirements to designs and code could be defined. Tool or matrix-based tracing would enhance the consistency and coverage checking process between requirements, designs, and code. Furthermore, tracing could improve current problems with an impact analysis of a change.

6.3 Case two - small organisation

The second case organisation develops data management solutions for embedded devices. The organisation employs around 100 people. One-fourth of these employees work in R&D. Development projects at the organisation are carried out at one site with an incremental software development process. The organisation has an ongoing software process improvement program that is based on ISO/IEC 15504 standard.

Requirements come from different sources such as customers, sales and marketing departments, standards, and product management. These requirements are translated into an external specification written in natural language. The external specification then serves as a contract between stakeholders and developers. The original stakeholder requirements are not documented in a separate requirements document. In case of a new feature, the external specification is transformed into more detailed software requirements.

The organisation's director of R&D served as source of information for the case study. In addition, documents such as software process descriptions, requirement specification templates, and review guidelines were examined.

6.3.1 Current state analysis

The case organisation's requirements documentation was sufficiently well managed. Documentation was seen as useful and guidelines for the documentation were used. Also templates for requirements specifications were available. The requirements were properly classified and included appropriate attributes. Requirements documentation also included verification criteria for necessary non-functional requirements.

Requirements implementation in the coding and testing activities was also well handled. The coding activity included review practices for certain parts of the code where requirements implementation needed to be evaluated. Competent developers ensured absorption of quality attributes into the code, such as portability. Maintainability of the code was achieved by following the organisation's coding standards. The testing activity included a set of testing tools

that automatically ensured verification of correct realisation of requirements. Sometimes, however, automatic testing was not possible. In such case, manual work and tracking of test cases to requirements was required. Tracking between requirements and test cases was being introduced more and more.

The organisation was not experiencing many problems with changing requirements. A product manager managed the change process and changes were handled at weekly meetings. The organisation had established requirements baselining practice in use and had a change approval body similar to the change control board. Also software configuration management practices were in use to control change. The organisation planned to introduce a tailored requirements management tool to enhance requirements management and implementation.

Prioritisation of requirements was seen as an important factor for the whole development process, but questions arose about the effectiveness of the current prioritisation process. Currently, requirements prioritisation with various stakeholders is performed at meetings and is mainly based on intuition. Whether all necessary requirements were covered and whether they were correctly graded was not so evident.

Review practices in the organisation were not fully taken into use. When reviews were performed, they tended to follow a more informal than formal process. Limited resources simply do not allow extensive check of consistency between requirements and other work products.

Other insufficiencies were observed with requirements tracing and architecture designing. It was not always clear how to trace lower-level requirements to test cases. Architecture designing was reported to be well performed, but it was mainly based on designers' expertise and architecture analysis against quality attributes was not commonly performed.

6.3.2 Improvement proposal

To improve current practice of requirements prioritisation, the organisation should take a prioritisation method into use. Obviously, the method will not solve all the prioritisation problems by itself, but it would give a more

rationalised basis for making priority decisions. One possible solution would be Wiegers' prioritisation method that is based on evaluating requirements priority from the viewpoint of different stakeholders and developers.

Another improvement area to consider would be the current practice of reviews. More formal inspections may remove defects more efficiently and enhance consistency between requirements and work products. Extensive reviews in all development activities, however, are not a reality in small organisations. Therefore, a root cause analysis must be done to locate where defects have infiltrated and to improve review practices in the most vulnerable development activities. Automation of certain reviews would minimise developers' involvement. For example, code reviews may be supplemented with a static code analysis tool such as QAC/++ that verifies portability of the code.

Other improvements may be accomplished by using the upcoming requirements management tool to trace requirements to test cases and by utilising architectural analysis methods more commonly. Linking test cases to requirements with the requirements management tool would make it possible to prove when all features have been covered in the testing activity. Although architecture designing was well performed, architecture trade-off and quality analysis methods could be used to enhance evaluation of architecture against quality attributes.

6.4 Applicability of the requirements implementation framework

After the current state analyses and the improvement proposals on requirements implementation for the case organisations were made, the organisations evaluated the significance of the given improvement proposal. The organisations assessed the proposal's individual improvement suggestions significance with a five-grade scale, where the lowest grade indicated an insignificant improvement and the highest grade a critical improvement. This feedback information was used to evaluate the requirements implementation framework's applicability in improving requirements implementation practices.

The improvement proposal for the first case organisation was considered to be very significant. A total of seven individual improvement suggestions were made. Two of these suggestions were considered critical. Both of these concerned system requirements development and documentation. Furthermore, three suggestions were considered important. These suggestions included the involvement of other disciplines in software requirements reviews, architectural analysis against requirements, and static code analysis. The rest of the suggestions - quality attribute quantification and post-traceability - were seen as less important and did not therefore require immediate actions. The results from the large organisation's feedback indicated that the framework was very useful in revealing weak spots and developing improvement suggestions.

The second organisation received total of five improvement suggestions. The organisation evaluated two suggestions to be important and the rest of the suggestions as less important. The current state with requirements prioritisation and work product inspections were considered to be important improvement areas. Static code analysis and architecture evaluation were considered less important improvements and their utilisation would require a closer study to see the real benefits. Finally, tracing test cases to requirements with the requirements management tool was not currently considered to be feasible.

The improvement proposal made for the large organisation was more successful compared to the proposal made for the small organisation. One of the reasons for this is the fact that the large organisation had much clearer system and hardware levels in its development life cycle. Therefore, improvement areas in the framework that concerned those levels were not that applicable in the small organisation. Furthermore, the large organisation with a rigid and document oriented development process had naturally more need for the elements presented in the framework. Nevertheless, the improvement suggestions made for the small organisation were also considered important and therefore the framework can be also regarded as applicable for analysing and improving small organisations' requirements implementation practices.

7. Conclusions

This chapter concludes the study of requirements implementation in embedded software development. First, the research is summarised by answering the research questions. Then, the significance of the results is discussed. Finally, further research possibilities based on this study are given.

7.1 Answers to the research questions

This research studied the means to improve requirements implementation in embedded software development. The goal of the research was to gather elements for effective requirements implementation from the literature and integrate these into a common framework. The framework could be then used in analysing and improving organisations' requirements implementation practices. The goal was achieved by answering these research questions as set out in the introductory chapter:

1. How can requirements be effectively implemented in embedded software development?
 - 1.1. What is the relationship between requirements and development activities?
 - 1.2. How should requirements be implemented during development?
 - 1.3. What kind of framework would help to analyse and improve requirements implementation practices?

The first sub-question was answered in Chapter 3 by discussing the impact of various requirements on different development activities. It was shown that requirements manifest themselves clearly in all development activities and that quality attributes in particular are problematic to implement and verify.

Chapter 3 also partly provided answers to the second sub-question by presenting requirements implementation methods for distinct development activities. The second sub-question was further covered in Chapter 4, where requirements implementation-supporting elements were discussed. These elements included

requirements change management, traceability, and consistency checking, which are all visible throughout the development.

In Chapter 5, the requirements implementation elements found in Chapters 3 and 4 were gathered into a common framework in order to answer the third sub-question. The developed requirements implementation framework was intended to be used as a means of analysing and improving embedded software organisations' capability to implement requirements effectively. This hypothesis was validated in Chapter 6 which discussed the case studies in the small and the large organisation.

The results from the case studies showed that the framework was indeed applicable in analysing and improving requirements implementation practices in an actual industrial environment. Most of the given improvement suggestions for the large organisation were considered to be important or even critical. Part of improvement suggestions made for the small organisation were also considered important.

7.2 Significance of the results

The main result of this study is the requirements implementation framework for embedded software. The framework is plainly a synthesis of previous research results and used practices in the industry. What, then, is so novel and non-obvious about their integration? The main contribution of the framework is that it covers relevant elements as extensively as possible. It also represents relationships among these elements to increase knowledge about their interdependencies. Other meaningful results from this study are the questionnaire, which can be used to clarify an organisation's current state of requirements implementation practices, in Appendix 1, and references to the concrete processes, methods, and tools for these practices in Appendix 2. These both can be used as a basis for software process improvement from the viewpoint of requirements implementation.

The developed framework's strength and at the same time its weakness is that it covers a great deal of different software development areas and even system engineering aspects. This caused the framework to evolve into a fairly general

model. One element of the framework, for example requirements change management, could have formulated its own framework. On the other hand, requirements implementation could have been examined solely from one software development activity's viewpoint such as designing or coding. This would have made possible more detailed analysis and improvement suggestions on more specific areas.

However, general models are also needed. As was shown, the elements of the framework affect each other from the system requirements level down to software testing level. Therefore, the developed framework could identify improvement areas from a large perspective. Indeed, an overall picture of the problem domain is needed to isolate specific improvement areas. What then separates the framework from known software process improvement reference models such as CMMI or ISO 15504? This study's framework should be used especially when requirements implementation causes problems in development activities. Therefore, it may be used to complement software process improvement efforts that are based on the reference models, as was shown in the case studies.

After the framework was elaborated, its applicability in actual software development environment was validated. The main problem with the validation was the low number of case studies. Is it legitimate to draw conclusions about the framework's applicability from only two case studies? Furthermore, a case study may contain so many organisation-specific factors that it may be impossible to derive generic results (Potts 1993).

The results from the two case studies indicated that the framework was indeed useful. Two different kinds of organisations - a large one and a small - were studied to augment the validity of the results. In addition to the missing practices found in the organisations, the framework also revealed many well performed elements. This also supports the claim that the framework includes essential elements for effective requirements implementation. However, more case studies would be needed to substantiate the results. Conducting such studies is by no means an easy task, because a proper case study requires a very close examination of the subject or phenomenon. In this study, it meant examining a significant number of software documents, preparing and analysing questionnaires, and interviewing software developers.

The developed framework provides the means to obtain an overall picture of embedded software development and locate possible improvement areas related to requirements implementation. Although the framework was intended for embedded software, it is also applicable to analysing non-embedded software development processes. In this case, system engineering and hardware related elements from the framework may be ignored.

7.3 Further research possibilities

The requirements implementation framework's usefulness was validated in embedded software environment by making a current state analysis and proposing improvement suggestions. However, implementation of the proposed suggestions and their possible benefits were not studied in this research. Such a study should be conducted to evaluate the actual value of the framework based improvement suggestions.

The developed framework was shown to be an effective foundation for improving a large organisation's requirements implementation practices. More research is needed to tailor the framework so that it is more capable of improving a small organisation's problems cost-effectively. Further studies are also needed to clarify what kinds of requirements implementation elements are essential to the development of non-embedded software such as information systems.

The results from the case studies implied that while software requirements were reasonably well handled, there were serious problems with system requirements. In order to improve this, the participation of software developers at the system engineering level should be studied more carefully. For example, software measurement information and prototypes may be used to evaluate development costs and rationalise system requirements allocation procedures.

References

Arnold, R.S. & Bohner, S.A. 1993. Impact Analysis - Towards a Framework for Comparison. Proceedings of the Conference on Software Maintenance 1993, 292–301.

Basili, V.W. 1996. The Role of Experimentation in Software Engineering: Past, Current, and Future. Proceedings of the 18th International Conference on Software Engineering, 1996, 442–449.

Boehm, B. W. 1988. A Spiral Model of Software Development and Enhancement. IEEE Computer, Vol. 21, no. 5, 1988, 61–72.

Chechik, M. & Gannon, J. 1995. Automatic Analysis of Consistency between Implementations and Requirements: a Case Study. Proceedings of the Tenth Annual Conference on Computer Assurance, 1995, 123–131.

Chechik, M. & Gannon, J. 2001. Automatic Analysis of Consistency between Requirements and Design. IEEE Transactions on Software Engineering. Vol. 27, no. 7, 2001, 651–672.

CMMI-SW. 2002. Capability Maturity Model® Integration for Software Engineering, Continuous Representation, Version 1.1. Available: <http://www.sei.cmu.edu/pub/documents/02.reports/pdf/02tr001.pdf> [Referenced 2.6.2003].

Crnkovic, I., Funk, P. & Larsson, M. 1999. Processing Requirements by Software Configuration Management. Proceedings of 25th EUROMICRO Conference, 1999. Vol. 2, 260–265.

Davis, A.M. 1988. A Comparison of Techniques for the Specification of External System Behaviour. Communications of the ACM, Vol. 31, no. 9, 1988, 1098–1115.

Davis, A.M. 1990. Software Requirements: Analysis and Specification. New Jersey, USA: Prentice-Hall.

- DOORS 2003. Telelogic products - DOORS. Available:
<http://www.telelogic.com/products/doorsers/doors> [Referenced 15.11.2003]
- Dorfman, M. 1997. Requirements Engineering. In Thayer, R.H. & Dorfman, M. (eds.) 1997. Software Requirements Engineering, 2nd ed. Los Alamitos, USA: IEEE Computer Society Press.
- Douglass, B.P. 2000. Real-Time UML, 2nd ed. Reading, USA: Addison-Wesley.
- Easterbrook, S. 2001. Lecture slides on Software Lifecycles. Available:
www.cs.toronto.edu/~sme/CSC444F/slides/L04-Lifecycles.pdf [Referenced 20.5.2003].
- Ebert, C. 1997. Dealing with nonfunctional requirements in large software systems. *Annals of Software Engineering* 3, Kluwer Academic Publishers, 367–395.
- Ernst, R. 1998. Codesign of Embedded Systems: Status and Trend. *IEEE Design & Test of Computers*, Vol. 15, no. 2, 45–54.
- Gajski, D.D. & Vahid, F. 1995. Specification and Design of Embedded Hardware-Software Systems. *IEEE Design & Test of Computers*, Vol. 12, no. 1, 53–67
- Gotel, O.C.Z. & Finkelstein, C.W. 1994. An Analysis of the Requirements Traceability Problem. *Proceedings of the First International Conference on Requirements Engineering*, 94–101
- Gross, D. & Yu, E. 2001. From Non-Functional Requirements to Design Through Patterns. *Requirements Engineering*, Vol. 6, no. 1, 18–36.
- Grünbacher, P., Egyed, A. & Medvidovic, N. 2001. Reconciling Software Requirements and Architectures: The CBSP Approach. *Proceedings Fifth IEEE International Symposium on Requirements Engineering*, 202–221.
- Gupta, P. & Cuning, S.J. & Rozenblit, J.W. 2001. Synthesis of High-Level Requirements Models for Automatic Test Generation. *Proceedings of the Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, 76–82.

Harker, S.D.P., Eason, K.D. & Dobson, J.E. 1993. The Change and Evolution of Requirements as a Challenge to the Practice of Software Engineering. Proceedings of the IEEE International Symposium on Requirements Engineering 1993, 266–272.

Hofmeister C., Nord, R. & Soni, D. 2000. Applied Software Architecture. Reading, USA: Addison-Wesley.

IEEE Std 830–1998. Recommended Practice for Software Requirements Specification. 1998. Institute of Electrical and Electronics Engineering, Inc.

Inverardi, P., Muccini, H. & Pelliccione, P. 2001. Checking consistency between architectural models using SPIN. Proceedings of International Conference on Software Engineering 2001 Workshop “From Software Requirements to Architectures” (STRAW'01), 62–67.

ISO/IEC 12207. 1995. Information technology - Software lifecycle processes. ISO/IEC.

ISO/IEC 15504. 1998. Information technology - Software process assesment - Part 5: An assesment model and indicator guidance. Draft technical report. ISO/IEC.

Järvinen, P. & Järvinen, A. 2000. Tutkimustyön metodeista. Tampere, Finland: Opinpaja.

Kandt, R.K. 2002. Software Configuration Management Principles and Best Practices. In: Oivo, M. & Komi-Sirviö, S. (eds.) 2002, Product Focused Software Process Improvement. 4th International Conference PROFES 2002, LNCS 2559, Springer-Verlag, 300–313.

Karsai, G., Sztipanovits, J., Ledeczi, A. & Bapty, T. 2003. Model-Integrated Development of Embedded Software, Proceedings of the IEEE, Vol. 91, no. 1, January, 145–164.

Knight, J.C. & Myers, E.A. 1993. An Improved Inspection Technique. Communications of the ACM, November 1993, Vol. 36, No. 11, 51–61.

Kotonya, G. & Sommerville, I. 1998. Requirements Engineering, Processes and Techniques. Chichester, England: John Wiley & Sons Ltd.

Kozlenkov, A. & Zisman, A. 2002. Are their Design Specifications Consistent with our Requirements? Proceedings of the IEEE Joint International Conference on Requirements Engineering 2002, 145 –154.

Lam, W. & Shankararaman, V. 1999. Requirements Change: a Dissection of Management Issues. Proceedings of 25th EUROMICRO Conference, 1999. Vol. 2, 244–251.

Lee, E.A. 2000. What's Ahead for Embedded Software? IEEE Computer, Vol. 33, no. 9, September 2000, 18–26.

Lindström, D.R. 1993. Five Ways to Destroy A Development Project. IEEE Software, Vol. 10, no. 5, September 1993, 55–58.

Liu, L. & Yu, E. 2001. From Requirements to Architectural Design - Using Goals and Scenarios. ICSE-2001 Workshop: From Software Requirements to Architectures. Available: <http://www.cs.toronto.edu/pub/eric/STRAW01-R2A.pdf> [Referenced 20.5.2003].

Logiscope. 2003. Telelogic Tau Logiscope. Available: <http://www.telelogic.com/products/tau/logiscope/> [Referenced 17.9.2003].

López, J.C., Hermida, R. & Geisselhardt, W. 1998. Advanced Techniques for Embedded Systems Design and Test. Dordrecht, The Netherlands: Kluwer Academic Publishers.

Mazza, C., Fairclough, J., Melton, B., De Pablo, D., Scheffer, A., Stevens, R., Jones, M. & Alvisi, G. 1996. Software Engineering Guides. Prentice Hall.

Mead, N.R. 1994. The Role of Software Architecture in Requirements Engineering. Proceedings of the First International Conference on Requirements Engineering 1994, 242.

MOOSE 2002a. Survey of Applicable Architectural Design and Analysis Methods. MOOSE Consortium.

MOOSE 2002b. Industrial Inventory Report. MOOSE Consortium.

Mäkäräinen, M. 2000. Software Change Management Processes in the Development of Embedded Software. VTT Publications 416. Espoo, Finland: Technical Research Centre of Finland.

O'Neal, J.S. & Carver, D.L. 2001. Analysing the Impact of Changing Requirements. Proceedings of IEEE International Conference on Software Maintenance, 2001, 190–195.

Olsson, T. & Grundy, J.C. 2002. Supporting Traceability and Inconsistency Management Between Software Artifacts. Proceedings of the IASTED International Conference on Software Engineering and Applications 2002, Boston, USA.

Olsson, T. & Runeson, P. 2002. Document Use in Software Development: A Qualitative Survey. Software Engineering, research and practise in Sweden 2002. Available: http://serg.telecom.lth.se/research/publications/docs/100_olsson_runeson_document_use.pdf. [Referenced: 5.6.2003].

Palmer, J.D. 1997. Traceability. In Thayer, R.H. & Dorfman M. (eds.) 1997, Software Requirements Engineering, Vol. 2. Los Alamitos, USA: IEEE Computer Society Press. 364–374.

Parviainen, P., Hulkko, H., Kääriäinen, J., Takalo, J. & Tihinen, M. 2003. Requirements Engineering, Inventory of technologies. VTT Publications 508. Espoo, Finland: Technical Research Centre of Finland.

Potts, C. 1993. Software-Engineering Research Revisited. IEEE Software. Vol. 10, no. 5, September 1993, 19–28.

Pressman, R.S. 2000. Software Engineering, A Practitioner's Approach, European Adaptation, 5th ed. Berkshire, England: McGraw-Hill.

Punter, T., Trendowicz, A. & Kaiser, P. 2002. Evaluating Evolutionary Systems. In Oivo, M. & Komi-Sirviö, S. (eds.) 2002, Product Focused Software Process Improvement. 4th International Conference PROFES 2002, LNCS 2559, Springer-Verlag, 258–271.

QAC/++. 2003. QAC/++. Available:
<http://www.programmingresearch.com/solutions/qac3.htm> [Referenced 17.9.2003].

Ramesh, B., Powers, T., Stubbs, C. & Edwards, M. 1995. Implementing Requirements Traceability: a Case Study. Proceedings of the Second IEEE International Symposium on Requirements Engineering 1995, York, England, 89–95.

Rational RequisitePro 2003. Rational RequisitePro - Product overview. Available: <http://www-306.ibm.com/software/awdtools/reqpro> [Referenced: 15.11.2003].

Robinson, W.N. 2002. Monitoring Software Requirements using Instrumented Code. Proceedings of the 35th Hawaii International Conference on System Sciences 2002, 3600–3609.

Robinson, W.N. & Pawlowski, S.D. 1999. Managing Requirements Inconsistency with Development Goal Monitors. IEEE Transactions on Software Engineering, Vol. 25, no. 6, 1999, 816–835.

Rodríguez-Dapena, P., Vardanega, T., Trienekens, J. & Brombacher, A. 2001. Nonfunctional Requirements as a Driving Force of Software Development. Software Quality Professional, Vol. 3, no. 4, 2001. Available: http://www.asq.org/pub/sqp/past/vol3_issue4/nonfunctional.html. [Referenced 24.6.2003].

Ronkainen, J., Taramaa, J. & Savuoja, A. 2002. Characteristics of Process Improvement of Hardware-Related SW. In Oivo, M. & Komi-Sirviö, S. (eds.) 2002, Product Focused Software Process Improvement. 4th International Conference PROFES 2002, LNCS 2559, Springer-Verlag, 247–257.

RTM 2003. Requirements & Traceability Management.

Available: <http://www.chipware.com/> [Referenced 15.11.2003].

Ruhe, G., Eberlein, A. & Pfahl D. 2003. Trade-off Analysis for Requirements Selection. *International Journal of Software Engineering and Knowledge Engineering*, Vol. 13, no. 4, 345–366.

Sangiovanni-Vincentelli, A. & Martin, G. 2001. Platform-Based Design and Software Design Methodology for Embedded Systems. *IEEE Design & Test of Computers*, Vol. 18, no. 6, 23–33.

Sawyer, P. & Kotonya, G. 2001. Software Requirements. In: Abran, A., Moore, J.W., Bourque, P. & Dupuis, R. (eds.) 2001. SWEBOK Trial Version 1.00. Los Alamitos, USA: IEEE Computer Society Press.

Shull, F., Rus, I. & Basili, V. 2000. Perspective-Based Reading: Techniques for Improving Requirements Inspections. *IEEE Computer*. Vol. 33, no. 7, 73–79.

Sommerville, I. & Sawyer, P. 1997. Requirements Engineering. A good practice guide. Chichester, England: John Wiley & Sons Ltd.

Stankovic, J.A. 1996. Real-Time and Embedded Systems. *ACM Computing Surveys*, Vol. 28, no. 1, 205–208.

Stevens, R., Jackson, K., Brook, P. & Arnold, S. 1998. Systems Engineering. Coping With Complexity. Hertfordshire, England: Prentice Hall.

Tahat, L.H., Vaysburg, B., Korel, B. & Bader, A.J. 2001. Requirement-Based Automated Black-Box Test Generation. *Proceedings of the 25th Annual International Computer Software and Applications Conference*, 489–495.

Taramaa, J., Khurana, M., Kuvaja, P., Lehtonen, J., Oivo, M. & Seppänen, V. 1998. Product-based Software Process Improvement for Embedded Systems. Proceedings of 24th Euromicro Conference, Vol. 2, 905–912.

Thayer, R.H. & Dorfman, M. 1997. Software Requirements Engineering, 2nd ed. Los Alamitos, USA: IEEE Computer Society Press.

Travassos, G.H., Shull, F., Fredericks, M. & Basili, V.R. 1999. Detecting Defects in Object Oriented Designs: Using Reading Techniques to Increase Software Quality. Proceedings of OOPSLA '99, 47–56.

Wieggers, K.E. 1999. Software Requirements. Redmond, USA: Microsoft Press.

Weber, M. & Weisbrod, J. 2002. Requirements Engineering in Automotive Development - Experiences and Challenges. Proceedings of IEEE Joint International Conference on Requirements Engineering, 331–340.

Zelkowitz, M.V. & Wallace, D.R. 1998. Experimental Models for Validating Technology. IEEE Computer, Vol. 31, no. 5, 23–31.

Appendix 1: Questionnaire on current state of requirements implementation practices

Software requirements analysis

Software requirements documentation

- What kind of classification for software requirements is used? (e.g. functional, interface, quality attributes, design constraints)
- Is there a requirements documentation policy available in the system or software level? (policy = e.g. procedures, guidelines, templates, standards, etc.)
- If a documentation policy doesn't exist, is it clear how requirements are documented?

Software requirements' attributes

- What kind of attributes are specified for software requirements? (e.g. id, priority, stability, rationale, source)
- Are there some requirements attributes missing from the organisation's current requirement specifications?
- If a priority attribute for a requirement is defined, who are involved in the prioritisation of requirements (e.g. customers, managers, developers, etc.)?

Software requirements specification's verification

- Are software requirements specifications reviewed? If yes, how (e.g. informal review, formal inspection)?
- If specifications are reviewed, who are involved in the reviews (e.g. managers, designers, coders, testers)?
- Are quality attributes (i.e. non-functional requirements such as reliability, usability, safety, maintainability) in the SRS verified? If yes, how?

Software design

Software designers' input for software requirements development

- Are software designers involved in development of software requirements? If yes, how?

Software architecture's evaluation against software requirements

- Is requirements trade-off analysis made during the architectural design? If yes, how?
- Is the software architecture evaluated against software requirements during the design? If yes, how?
- Is the architectural design reviewed? If yes, is the implementation of software requirements (i.e. how the requirements are actually realised) in the architecture evaluated?
- If requirements implementation is evaluated in the review, how (e.g. informal review, formal inspection)?

Detailed designs' evaluation against software requirements

- Are detailed designs evaluated against software requirements during the design? If yes, how? If not, how is it known that requirements are correctly realised in the detailed designs?
- Are detailed designs reviewed? If yes, how (e.g. informal review, formal inspection) and is the requirements implementation evaluated?

Software coding

Software code's evaluation against software requirements

- Is code reviewed? If yes, how (e.g. informal review, formal inspection) and is the requirements implementation evaluated?
- Is absorption of quality attributes into the code verified? If yes, how (e.g. how is maintainability, modifiability, portability, efficiency of code checked)?

Software testing

Test case development

- When begins the development of test cases for software level testing?
- Are testers included in software requirements development?
- Is unfeasibility of certain requirements found while developing test cases?

Validation criteria for requirements in the SRS

- Are there specific validation criteria for requirements in the SRS? If not, are there problems testing certain requirements?
- Are there problems testing quality attributes? If yes, with what kind of quality attributes?

Evaluation of test cases and plans against software requirements

- Is the consistency between test cases and software requirements ensured? If yes, how? If not, are there problems with indistinct test cases?
- Is the coverage of all requirements within test cases ensured? If yes, how? If not, are there problems later with untested requirements?

Requirements change management

Requirements change management

- Are there problems in development activities (i.e. requirements analysis, design, coding and testing) with changing requirements?
- What kind of requirements change management processes, methods, tools are in use? How are they related to general change management?
- Are software requirements baselined? If not, are there problems with unstable requirements?
- Is a Change Control Board, Software Configuration Management, or test case management involved in requirements change management? If yes, how?

Requirements change into software requirements specification

- Is there a specific procedure for implementing a change into the SRS that comes from the system or hardware level? If not, are there problems related to implementing a change?
- Is there a specific procedure for implementing a change into the SRS that comes from a software development activity? If not, are there problems related to implementing a change?

Requirements changes impact to designs, code, and test cases

- Is there a change procedure for handling requirements change into software designs, code, and test cases? If not, are there problems implementing change?
- Is there a procedure for impact analysis of requirements change ? (e.g. process, guidelines, templates, etc. for analysing which components are affected and how much a change costs)
- To which work products is the impact of a requirement change analysed? (e.g. to architecture, detailed designs, code, test cases)
- Are there problems while performing impact analysis? If yes, what kind of problems?

Requirements tracing

- Are requirements traced from system level to software requirements? If yes, how (tools, matrices)?
 - from software requirements to architectures? If yes, how?
 - from software requirements to detailed designs? If yes, how?
 - from software requirements to code? If yes, how?
 - from software requirements to test cases? If yes, how?
- If requirements are traced, is there a requirements traceability policy in software level? (the policy may inform developers what to trace, why, and how)
- If requirements are traced, do developers have "motivation" to trace? (i.e. do they know why tracing is needed, and what are the benefits)

Unrealised requirements

- Is there a procedure for handling unrealised requirements in development activities? If not, how are unrealised requirements handled?
- Are change management and CCB involved in a clarification of an unrealised requirement?

Appendix 2: Requirements implementation practices

The following list summarises requirements implementation practices and related sample methods, techniques, and tools referenced to in this study.

Development activity	Practice	Sample methods, techniques, and tools
Software requirements analysis	Problem analysis and product description	Structured analysis, object-oriented analysis, formal methods, data flow diagrams, state-transition diagrams, use cases (Parviainen et al. 2003)
	Define attributes	Priority (QFD, Wiegers (1999)), Attributes (Sawyer & Kotonya 2001)
	Verification	Reviews, Formal inspection, Perspective-Based Reading (Shull et al. 2000)
Software design	Requirements trade-off analysis	ATAM, SBAR, AQA (MOOSE 2002a)
	Consistency check	CBSP (Grünbacher et al. 2001), GRL & UCM (Liu & Yu 2001), SCR & PDL (Chechik & Gannon 2001), Inspection (Travassos et al. 1999)
	Quality attribute verification	Prometheus (Punter et al. 2002)
Coding	Consistency check	Instrumented code (Robinson 2002), Analyzer (Chechik & Gannon 1995)
	Quality attribute verification	Phased inspection (Knight & Myers 1993), QAC/++ (2003), Logiscope (2003), Prometheus (Punter et al. 2002)
Testing	Consistency check	Automatic Test Scenario Generation (Gupta et al. 2001), Requirement-based Automatic Black-Box

		Testing (Tahat et al. 2001)
Throughout development	Change management	Olsen's change management model, Spiral-like change management model, and Generic change management process model (Parviainen et al. 2003)
	Requirements management	Rational RequisitePro (2003), DOORS (2003)
	Tracing	Matrices (Parviainen et al 2003), DOORS (2003), RTM (2003)
	Impact analysis	Quality Factor Deployment, requirements inspections, viewpoint analysis, trade-off analysis (Lam & Shankararaman 1999), Wiegers' templates (Wiegers 1999)

Author(s) Jääliinoja, Juho			
Title Requirements implementation in embedded software development			
Abstract Development of correct requirements at the beginning of a software project is considered an important precondition for successful software development. Moreover, implementing these requirements correctly during the software development is arguably just as important. Rigorous implementation of requirements in embedded software development is especially critical, since requirements affect both software and hardware. The goal of this research is to identify elements for effective requirements implementation in embedded software development. A conceptual-theoretical research approach is applied to analyse previous research on requirements implementation and to construct a new theory which integrates requirements implementation related elements into a holistic framework. These elements include requirements implementation processes, methods, and roles. The developed framework describes relations among these elements and furthermore their relation to software development activities. The framework can be used as a basis for improving software development areas that are related to requirements implementation. To validate the feasibility of the developed framework, two case studies were carried out within embedded software development organisations. The validation was conducted by making a current state analysis and by suggesting improvements based on the developed requirements implementation framework. The results from the case studies indicated that the framework was a useful foundation for improving the organisations' requirements implementation practices.			
Keywords software process improvement, software requirements analysis, embedded systems			
Activity unit VTT Electronics, Kaitoväylä 1, P.O. Box 1100, FIN-90571 OULU, Finland			
ISBN 951-38-6370-0 (URL: http://www.vtt.fi/inf/pdf/)		Project number E2SU00054	
Date April 2004	Language English	Pages 82 p. + app. 7 p.	Price
Name of project MOOSE (Software engineering methodologies for embedded systems)		Commissioned by	
Series title and ISSN VTT Publications 1455-0849 (URL: http://www.vtt.fi/inf/pdf/)		Sold by VTT Information Service P.O.Box 2000, FIN-02044 VTT, Finland Phone internat. +358 9 456 4404 Fax +358 9 456 4374	

VTT PUBLICATIONS

- 509 Sallinen, Mikko. Modelling and estimation of spatial relationships in sensor-based robot workcells. 2003. 218 p.
- 510 Kauppi, Ilkka. Intermediate Language for Mobile Robots. A link between the high-level planner and low-level services in robots. 2003. 143 p.
- 511 Mäntyjärvi, Jani. Sensor-based context recognition for mobile applications. 2003. 118 p. + app. 60 p.
- 512 Kauppi, Tarja. Performance analysis at the software architectural level. 2003. 78 p.
- 513 Uosukainen, Seppo. Turbulences as sound sources. 2003. 42 p.
- 514 Koskela, Juha. Software configuration management in agile methods. 2003. 54 p.
- 516 Määttä, Timo. Virtual environments in machinery safety analysis. 2003. 170 p. + app. 16 p.
- 515 Palviainen, Marko & Laakko, Timo. mPlaton - Browsing and development platform of mobile applications. 2003. 98 p.
- 517 Forsén, Holger & Tarvainen, Veikko. Sahatavaran jatkojalostuksen asettamat vaatimukset kuivauslaadulle ja eri tuotteille sopivat kuivausmenetelmät. 2003. 69 s. + liitt. 9 s.
- 518 Lappalainen, Jari T. J. Paperin- ja kartonginvalmistusprosessien mallinnus ja dynaaminen reaaliaikainen simulointi. 2004. 144 s.
- 519 Pakkala, Daniel. Lightweight distributed service platform for adaptive mobile services. 2004. 145 p. + app. 13 p.
- 520 Palonen, Hetti. Role of lignin in the enzymatic hydrolysis of lignocellulose. 2004. 80 p. + app. 62 p.
- 521 Mangs, Johan. On the fire dynamics of vehicles and electrical equipment. 2004. 62 p. + app. 101 p.
- 522 Jokinen, Tommi. Novel ways of using Nd:YAG laser for welding thick section austenitic stainless steel. 2004. 120 p. + app. 12 p.
- 523 Soininen, Juha-Pekka. Architecture design methods for application domain-specific integrated computer systems. 2004. 118 p. + app. 51 p.
- 525 Mäntyniemi, Annukka, Pikkarainen, Minna & Taulavuori, Anne. A Framework for Off-The-Shelf Software Component Development and Maintenance Processes. 2004. 127 p.
- 526 Jääliñoja, Juho. Requirements implementation in embedded software development. 2004. 82 p. + app. 7 p.

Tätä julkaisua myy
VTT TIETOPALVELU
PL 2000
02044 VTT
Puh. (09) 456 4404
Faksi (09) 456 4374

Denna publikation säljs av
VTT INFORMATIONSTJÄNST
PB 2000
02044 VTT
Tel. (09) 456 4404
Fax (09) 456 4374

This publication is available from
VTT INFORMATION SERVICE
P.O.Box 2000
FIN-02044 VTT, Finland
Phone internat. +358 9 456 4404
Fax +358 9 456 4374
